UNIVERSITÉ DE SHERBROOKE
Faculté de Génie
Département de génie électrique et de génie informatique

OUTIL DE COMMUNICATION POUR LES SYSTÈMES MULTIAGENTS

Mémoire de maîtrise es sciences appliquées
Spécialité: génie informatique

Noomen HAMZA

Sherbrooke (Québec), Canada Juillet 1998

À mes parents, à mes frères et soeurs,

je dédie ce mémoire...

RÉSUMÉ

Les recherches dans le domaine des systèmes multiagents étudient la manière de résoudre un problème complexe avec un certain nombre d'entités plus ou moins intelligentes. Ces entités, appelées aussi agents, coopèrent par communication pour aboutir à une solution d'un problème. Ces recherches se penchent aussi sur la manière de coordonner le comportement intelligent d'un ensemble d'agents selon des lois sociales.

À la base de la coopération entre agents, il est nécessaire que chaque agent ait des facilités de communication. Le projet associé à ce mémoire consiste à la conception et à la mise en œuvre d'un outil de communication entre agents dans un système multiagents. L'outil réalisé se présente sous forme d'une librairie de classes écrites en Java, permettant à des applications multiagents d'échanger, sans difficultés, des données entre ces différents agents. Ce développement sert à faire tomber les limitations des outils de développement de systèmes multiagents qui ne facilitent pas la tâche des chercheurs pour la validation des résultats théoriques de leurs recherches.

Pour illustrer et valider le bon fonctionnement de l'outil, nous avons conçu et mis en œuvre une application multiagents. Cette application consiste à simuler le nettoyage d'une surface avec obstacles par un ensemble de robots-aspirateurs communicants.

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui m'ont aidé, de près ou de loin, durant toutes mes années d'études, en particulier mes parents pour leur soutien moral et matériel et mon frère Radwen pour son encouragement. Je tiens à remercier aussi M. Ruben Gonzalez-Rubio pour ses conseils et ses services en tant que directeur de recherche ainsi que tout le personnel du Département de génie électrique et de génie informatique pour sa disponibilité.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LEXIQUE

IA : intelligence artificielle.

IAD : intelligence artificielle distribuée.

SMA : système multiagents.

OCA : outil de communication entre agents.

EGSMA : environnement générique pour le développement de systèmes multiagents.

Chapitre 1

INTRODUCTION

La naissance de l'intelligence artificielle (IA) dans les années cinquante a provoqué une grande révolution dans le monde de l'informatique. Ainsi, depuis ce temps-là, le nombre d'applications, le volume et la complexité de celles-ci n'ont pas cessé d'augmenter. Devant cette progression rapide des applications, l'IA classique, avec son principe de centralisation de l'expertise au sein d'un même système, se trouve très limitée surtout quand il s'agit de manipuler plusieurs expertises dans un même système.

Les systèmes multiagents (SMA) sont apparus dans les années soixante-dix [8] pour nous offrir les moyens pour gérer et distribuer plusieurs domaines d'expertise dans un même système. Un SMA est défini comme étant un ensemble d'entités ou d'agents plus ou moins intelligents, capables de communiquer et d'agir avec ses propres moyens et connaissances. Un agent peut, en cas de besoin, demander la collaboration d'un autre agent, pour exécuter sa propre tâche ou pour exécuter une tâche qui lui a été affectée. Les agents convergent alors vers un résultat final par coopération et compétition entre eux. Toutes les actions d'interactions

d'un agent (coopération, planification, etc.) sont basées sur la communication entre agents.

Dans ce mémoire, nous présentons un outil de communication entre agents pour les SMA. L'outil qu'on propose, nommé OCA, se présente sous forme d'une librairie de classes développées en Java, offrant aux programmeurs de SMA les fonctions de bases pour permettre à un ensemble d'agents de communiquer entre eux sans difficultés. L'outil OCA permet à un ensemble d'agents de communiquer par envoi de messages asynchrones. Comme le développement de OCA a été fait en Java, ce dernier sera portable sur des différents systèmes d'exploitation. Ceci nous permet d'implanter des agents sur un environnement hétérogène. Cet environnement pourrait être formé par un ensemble de machines distinctes, gérées par des systèmes d'exploitation différents et liées par un réseau de communication public ou local.

1.1 Cadre du projet de recherche

Les travaux de développement de OCA font partie intégrante d'un autre projet de recherche dont les travaux se déroulent au Département de génie électrique et de génie informatique de l'Université de Sherbrooke. Ce projet consiste à la conception et à la réalisation d'un environnement générique pour le développement d'applications basées sur les systèmes multiagents (EGSMA). Dans ce projet, une structure d'agent à quatre niveaux à été proposée. Les deux niveaux les plus internes constituent une interface de communication pour un agent. Cette interface offre à un agent les services de base de communication pour qu'il puisse interagir avec d'autres agents au sein d'un SMA. Notre participation dans ce projet consistait à la conception et à la réalisation de cette interface de communication.

1.2 Présentation des objectifs

Pour la mise en œuvre de l'outil OCA, nous avons fixé quelques objectifs à atteindre :

- La complétude : L'outil doit répondre à tous les besoins en communication entre différents types d'agents pour que ces derniers puissent accomplir leurs tâches d'interaction de façon convenable et sans difficulté.
- La portabilité : L'outil doit permettre la communication entre agents exécutés dans un environnement hétérogène. L'outil doit donc être le plus possible indépendant de l'environnement utilisé (matériel et logiciel).
- La simplicité : L'outil doit être simple et facile à l'utilisation. Ceci permet d'accélérer la tâche de programmation pour le développement multiagents et de réduire les délais de mise sur le marché des applications multiagents.
- L'efficacité : La communication est vitale pour un agent dans une société d'agents. En effet, le comportement d'un SMA (coopération, coordination, planification, etc.) est basé sur la communication. D'où l'importance d'avoir une interface de communication optimisée sur le plan temps d'exécution et ressources utilisées.
- L'évolution : L'outil doit être flexible pour toute opération de maintenance, d'amélioration ou d'ajout de nouvelles fonctionnalités.

1.3 Réalisation

Pour la mise en œuvre de l'outil OCA, nous avons développé une librairie de classes écrites en Java. Dans cette librairie on trouve sept classes scindées en deux catégories :

- Les classes publiques : dans cette catégorie on trouve deux classes :
  - La classe `agentCommunicant` : c'est la classe principale de l'outil OCA vis-à-vis de l'utilisateur. Elle représente au fait un modèle d'agent communicant intégrant une interface de communication offrant aux utilisateurs plusieurs fonctions d'interaction pour un agent. Les principaux types de fonctions qu'on trouve dans cette classe sont :
    - Les fonctions de gestion de dialogues entre agents (établir un dialogue, suspendre un dialogue, rompre un dialogue, etc.)
    - Les fonctions d'échange de messages entre agents (émettre un message, diffuser un message, recevoir un message, etc.)
    - Les fonctions de gestion d'une boîte aux lettres (récupérer un nouveau message, attendre la réception d'un message, consulter un message, etc.)
    - Les fonctions de traitement des événements externes (la signalisation d'une rupture de dialogue, la signalisation de réception d'un accusé de réception relative à un message émis, la signalisation d'une demande d'établissement de dialogue, etc.)
  - La classe `mail` : cette classe représente une enveloppe dans laquelle on trouve l'information utile échangée entre les différents agents. Dans cette classe on trouve aussi d'autre données telles que la destination du message, la priorité du message, le type d'émission (avec ou sans accusé de réception), etc.

- Les classes privées : dans cette catégorie les classes sont transparentes vis-à-vis de l'utilisateur. En effet, nous avons créé ces classes pour permettre le traitement multi-tâche au sein de la classe `agentCommunicant`. Chaque classe est dédiée pour un traitement spécifique (émission de messages, réception de messages, détection des demandes de dialogue, etc.). Les traitements de ces classes s'exécutent en concurrence avec ceux de la classe `agentCommunicant`.

L'utilisation de OCA pour la mise en œuvre d'un agent communicant consiste à :

1. Rejoindre la librairie de classes de l'outil OCA à celle de Java.
2. Créer une classe dérivée de la classe `agentCommunicant`.
3. Rajouter à cette classe les fonctions nécessaires pour définir le comportement d'agent en faisant appel aux fonctions prédéfinies dans classe `agentCommunicant`.
4. Réécrire les fonctions de traitement des événements externes pour leurs traitements.

Pour illustrer le bon fonctionnement et l'utilité de l'outil OCA, nous avons proposé une application multiagents qui simule le comportement d'un ensemble de robots-aspirateurs travaillant sur une surface poussiéreuse avec des obstacles. L'objectif général de ces robots est de nettoyer la surface en faisant un minimum de déplacements. L'application a été développée en Java et OCA. Les résultats de la simulation étaient bien satisfaisants. Ceci nous a démontré en partie le bon fonctionnement et l'utilité de l'outil OCA.

1.4 Organisation du mémoire

Dans la suite de ce mémoire, nous exposerons d'abord dans le chapitre 2 l'état de l'art des SMA en expliquant les différents comportements et architectures qu'une société d'agents peut avoir et le rôle important que joue la communication dans celle-ci pour une résolution ou une exécution coopérante. Nous présentons aussi dans ce chapitre quelques outils de développement de SMA avec une étude comparative. Ensuite, nous proposons dans le chapitre 3 les spécifications d'un nouvel outil de communication pour les SMA. Le chapitre 4 sera consacré pour les détails de la conception et de la mise en œuvre de cet outil. Par la suite, pour illustrer l'utilité et le bon fonctionnement de cet outil nous proposons, dans le chapitre 5, une application multiagents développée en Java et OCA. Finalement, en guise de conclusion, nous soulignons dans le chapitre 6 l'utilité de notre outil dans le domaine multiagents et les perspectives prévues pour ce dernier. Nous expliquons ainsi les futures améliorations que nous pourrions apporter à notre outil pour qu'il réponde davantage aux besoins des applications multiagents qui ne cessent d'évoluer et de se compliquer de plus en plus.

Chapitre 2

L'état de l'art des SMA

2.1 Introduction

L'évolution des applications de l'IA dans des domaines complexes et hétérogènes tels que l'aide à la décision, la reconnaissance et la compréhension des formes, la conduite des processus industriels, etc., a montré les limites de l'approche classique de l'IA qui s'appuie sur une centralisation de l'expertise au sein d'un système unique.

Les travaux menés au début des années soixante-dix sur la concurrence et la distribution ont contribué à la naissance d'une nouvelle discipline : l'Intelligence Artificielle Distribuée (IAD) [8] [31] [20]. L'IAD a pour but de remédier aux insuffisances de l'approche classique de l'IA en proposant la distribution de l'expertise sur un groupe d'agents devant être capables de travailler et d'agir dans un environnement commun et de résoudre les conflits éventuels. D'où la naissance de notions nouvelles en IA, telles que la coopération, la coordination d'actions et la négociation. L'IAD est définie dans [31] comme étant la branche de l'IA qui s'intéresse à la

modélisation d'un comportement «intelligent» (jusqu'ici c'est la définition de l'IA classique) par la coopération entre un ensemble d'agents.

L'IAD conduit entre autres à la réalisation des systèmes dits «multiagents» qui permettent de modéliser le comportement d'un ensemble d'entités plus ou moins expertes, plus ou moins organisées. Ces entités, appelées aussi agents, disposent d'une certaine autonomie et sont immergées dans un environnement dans lequel et avec lequel elles interagissent. La résolution d'un problème se fait alors par coopération ou compétition entre agents.

L'efficacité et la problématique de tels systèmes résident dans le choix de méthodes de coordination ou de communication entre agents. Celles-ci définissent la façon dont vont interagir les parties du système pour résoudre le problème posé. Chacune des méthodes possède des avantages et des inconvénients qui font qu'elles sont plus ou moins adéquates selon le type de problème posé.

Plusieurs recherches menées sur ce domaine ont abouti à des résultats plus ou moins satisfaisants suivant les cas d'applications. Ces recherches ont portées sur quatre axes :

- Le degré de coopération entre agents ;
- La communication entre agents ;
- Les structures et les architectures des SMA ;
- Les formalismes de représentation (par objet, logique du premier ordre, logique floue, réseau de neurones, etc.)

2.2 Problématique de l'IAD

Selon Labidi et Lejouad [33], les problèmes que l'IAD s'attache à résoudre peuvent être divisés en deux classes. La première concerne les problèmes classiques de l'IA qui ont pris une nouvelle dimension dans le contexte multiagents. La seconde classe regroupe les nouveaux problèmes proprement liés au thème de l'IAD. Dans la première classe on traite :

- La modélisation de la connaissance et le problème de sa répartition sur les différents agents : Comment formuler, décomposer, allouer des problèmes et synthétiser les résultats d'un groupe d'agents ?
- Les problèmes de génération de plan d'action en prenant en considération la présence d'autres agents ;
- La gestion des conflits entre les agents (points de vue différents des agents) et le maintien de la cohérence des décisions et des plans d'action ;
- Le problème de la communication : Comment permettre la communication et l'interaction entre les agents ? Quel langage et quel protocole faut-il employer ? Une communication dans les univers multiagents n'est plus une simple tâche d'entrée-sortie. Elle doit être modélisée comme un acte pouvant influer sur l'état des autres agents.

Les problèmes de la seconde classe peuvent être divisés en deux :

- Les problèmes spécifiques au groupe d'agents, qui portent sur l'organisation, l'architecture de l'ensemble des agents et les paradigmes de coopération et d'action ;

- Les problèmes liés au comportement d'un agent au sein d'un groupe. On s'intéresse aux capacités sociales d'un agent : le partage des tâches, le partage des ressources, le raisonnement sur les autres agents (pouvoir modéliser leurs connaissances et être en mesure de connaître leurs plans d'action et de raisonner en fonction de ces plans).

D'autres thèmes de recherche sont présents dans le contexte multiagents, à savoir le raisonnement temporel, le raisonnement hypothétique, la représentation de la connaissance imprécise, etc.

2.3 Les systèmes multiagents

Contrairement à l'approche centralisée de l'IA, l'IAD vise la distribution de l'expertise sur un ensemble de composants qui communiquent pour atteindre un objectif global (élaboration d'un diagnostic, résolution d'un problème, etc.). Toutefois, cette approche nécessite la division du problème en sous-problèmes. Selon Ginsberg [25], cette hypothèse n'est pas toujours vraie, car beaucoup de problèmes ne peuvent être partitionnés de cette manière.

Une extension des systèmes d'IAD est proposée par Konologie dans [32] : dans le but d'une coopération effective, les composants doivent être capables de raisonner sur les connaissances et les capacités d'autres composants. Pour ce faire, ils doivent être dotés de capacités de perception et d'action sur leur environnement et doivent posséder une certaine autonomie de comportement. On parle alors d'agents et par conséquent de SMA.

Les SMA se caractérisent alors par l'autonomie et l'intelligence des composants impliqués. Toutefois, un agent ne dispose pas d'une vision globale de son environnement [16].

2.3.1 Concept d'agent dans un SMA

Selon Ferber et Ghallab [20], un agent est défini comme une entité (physique ou abstraite) capable d'agir sur elle-même et son environnement, disposant d'une représentation partielle de cet environnement, pouvant communiquer avec d'autres agents et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents. Les agents ont deux tendances : une tendance sociale tournée vers la collectivité (les mécanismes et connaissances associés concernent les activités du groupe) et une tendance individuelle avec des mécanismes et des connaissances contenant les règles de fonctionnement interne de l'agent. Un agent peut avoir un rôle, une spécialité, des objectifs, des croyances, des capacités décisionnelles, des capacités de communication et éventuellement des capacités d'apprentissage.

2.3.2 Architecture d'agent

La figure 2.1 décrit l'architecture globale d'un agent. C'est une synthèse des architectures d'agent décrites dans la littérature. On distingue essentiellement :

- Le savoir-faire : Le savoir-faire est une interface permettant la déclaration des connaissances et des compétences de l'agent. Il permet la sélection des agents à solliciter pour une tâche donnée. Il n'est pas nécessaire, mais il est très utile pour améliorer les performances du système, quel que soit le mode de coopération utilisé.

- Les croyances : Dans un univers multiagents, chaque agent possède des connaissances sur lui-même et sur les autres. Ces connaissances ne sont pas nécessairement objectives ; On parle alors de croyances d'un agent. Les logiques des connaissances et des croyances

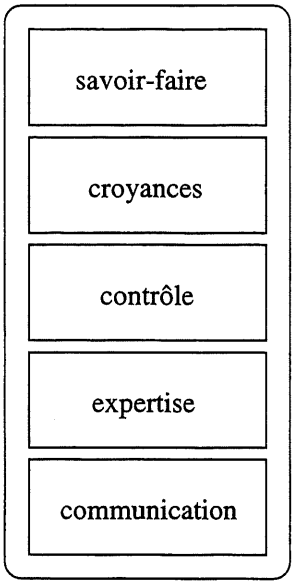

Figure 2.1 - *Structure d'un agent*

s'intéressent à la formalisation de telles connaissances considérées comme incertaines. Selon Halpern et Moses [29], cette formalisation est la base de la conception de tout SMA puisqu'elle détermine en grande partie le comportement «intelligent» des agents.

- Le contrôle : La connaissance de contrôle dans un agent est représentée par les buts, les intentions, les plans et les tâches qu'il possède.

- L'expertise : C'est la connaissance sur la résolution de problème. Pour un système expert utilisant le formalisme de règle, par exemple, cette connaissance correspond à sa base de règles.

- La communication : L'agent doit posséder un protocole de communication lui permettant d'interagir avec les autres agents pour une bonne coopération et une bonne coordination d'action. D'autres connaissances de communication peuvent être disponibles, par exemple les connaissances sur les réseaux de communication (tous les agents ne

sont pas forcément en liaison directe). À ces différents types de connaissances, on peut ajouter la connaissance liée au mode de la coopération : fonctionnement en mode appel d'offre, en mode compétition ou en mode commande.

2.3.3 Fonctionnement d'agent

Dans ce paragraphe, nous présentons notre vision de l'architecture fonctionnelle d'un agent et nous illustrons les détails de son fonctionnement. Cette architecture est illustrée par la figure 2.2. Les agents sont immergés dans un environnement dans lequel et avec lequel ils interagissent. D'où leur structure autour de trois fonctions principales : percevoir, décider et agir. Parmi les sous-fonctions importantes d'un agent, on peut citer : la détection de conflit, la révision des croyances, la coopération (négociation, coordination), l'apprentissage, etc. Toutes les fonctionnalités ne sont pas représentées dans la figure.

Un agent a la possibilité d'acquérir des connaissances sur l'environnement externe (perception). Il a aussi des capacités d'interaction avec les autres agents en communiquant avec ces derniers. En fonction des connaissances et croyances dont il dispose et des buts qu'il se fixe à la suite d'une perception ou d'une interaction avec le monde extérieur, l'agent doit élaborer un plan d'action. Pour ce faire, il doit décider quel serait le but à retenir et à satisfaire en premier. Ensuite, il doit planifier en fonction de ce but et passer à l'exécution. Ces deux derniers processus doivent être alternés du fait du caractère dynamique des environnements multiagents [33].

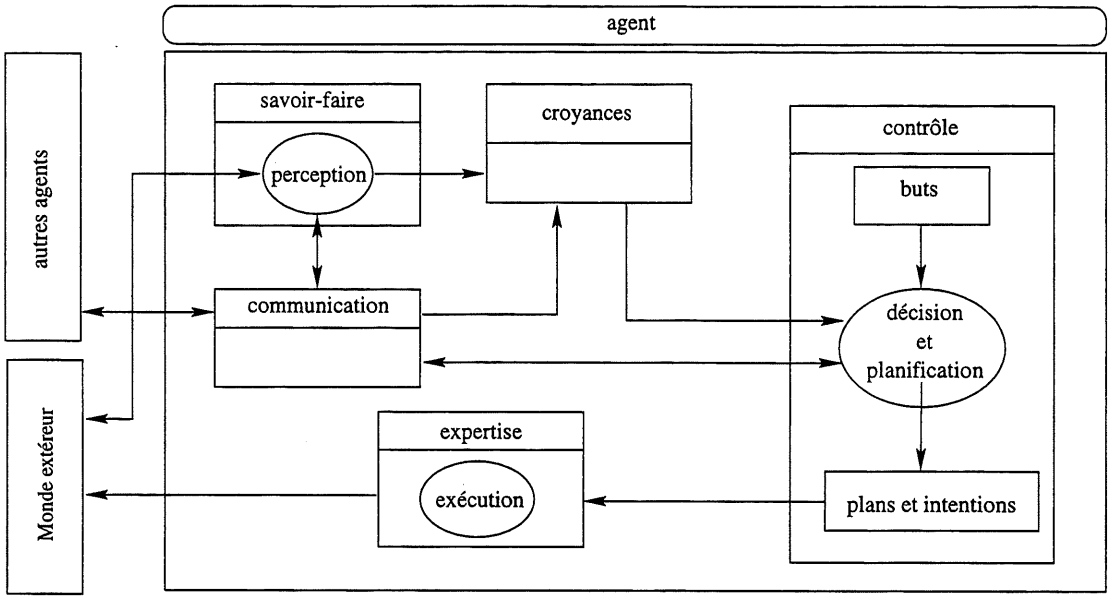

Figure 2.2 - *Fonctionnement d'un agent*

Perception

Les connaissances ou les croyances d'un agent ont plusieurs origines :

- Le savoir initial de l'agent ;
- La perception de soi (perception proprioceptive) et du monde externe (perception extéroceptive) ;
- La communication avec les autres agents.

Généralement, les informations issues de la perception et du savoir initial de l'agent sont considérées comme des connaissances certaines, puisqu'elles n'ont subi aucune mise à jour. Pour leur part, les connaissances provenant des autres agents sont considérées incertaines, puisqu'elles évoluent sans que l'agent en soit forcément informé. On peut, pour cela, asso-

cier à chaque connaissance son origine afin d'en évaluer la crédibilité et d'en permettre la vérification.

Communication

La communication est essentielle pour un agent en ce qui concerne toute interaction avec d'autres. La communication permet à un agent d'enrichir ou de réviser ces connaissances, de remettre en cause ses plans d'action, de coopérer ou de négocier avec d'autres agents, etc.

Prise de décision

Durant son exécution, un agent se fixe un certain nombre de buts, suite à ses observations et ses interactions avec le monde (perception, communication, négociation). Il se trouve donc confronté au problème de la sélection du but prioritaire et de l'action qui permet d'atteindre chaque but. Face à de telles situations, l'agent analyse les différentes alternatives en termes d'utilité (quel avantage l'agent pourrait en retirer) et d'incertitude (quelle chance a l'action de fournir le résultat attendu). Parmi les techniques utilisées pour la résolution de tels conflits, on peut citer l'utilisation de la notion de carte cognitive (Cognitive Map) [3] qui est un réseau qualitatif exprimé en terme d'influences (positives ou négatives) et reliant les buts et leur utilité. Le but à retenir sera celui qui présente le plus d'influences positives parmi les buts les plus interdépendants. La prise de décision est l'une des caractéristiques des agents rationnels, l'agent tiendra compte de ses croyances pour faire son choix.

Planification

La planification dans les systèmes d'IA classique repose sur l'hypothèse d'un univers statique (seules les actions du planificateur sont prises en compte), cet aspect est incompatible avec l'approche multiagents [14].

Les SMA , en revanche, offrent des possibilités de négociation autorisant une gestion locale des conflits et une planification dynamique. En effet, du fait de l'intervention d'autres agents, un plan peut être remis en cause. L'agent doit, pour cela, alterner planification et exécution et réviser des parties de son plan.

La planification dans les SMA est une planification distribuée. En effet, il n'existe pas de plan global. Chaque agent construit son propre plan en coordonnant avec les autres agents (cas d'agents coopératifs). Certains SMA présentent une planification centralisée, un organe central se chargera de la gestion des conflits et de l'élaboration d'un plan global.

Exécution

Finalement, l'exécution consiste à la mise en œuvre des plans générés par l'agent. L'exécution de ces plans est basée sur l'expertise de l'agent.

2.3.4 Caractéristique d'un agent

L'avancement des travaux en IAD et SMA a conduit les chercheurs à définir non seulement la notion d'agent, mais aussi quelques-unes de ses caractéristiques :

- Intentionnalité : Un agent intentionnel est un agent guidé par ses buts. Une intention est la déclaration explicite des buts et des moyens d'y parvenir. Elle exprime donc la volonté d'un agent d'atteindre un but ou d'effectuer une action [41] [46] [40].

- Rationalité : Bond et Gasser [37] définissent un agent rationnel comme étant un agent qui suit le principe suivant (dit principe de rationalité): «Si un agent sait qu'une de ses actions lui permet d'atteindre un de ses buts, il la sélectionne». Les agents rationnels disposent de critères d'évaluation de leurs actions et sélectionnent, selon ces critères, les meilleures actions qui leur permettent d'atteindre le but. De tels agents sont capables de justifier leurs décisions. La notion de rationalité se rapporte au comportement cognitif de l'agent. Ce terme qualifie l'utilisation des ressources par l'agent [37].

- Engagement : La notion d'engagement est l'une des qualités essentielles des agents coopératifs. Un agent coopératif planifie ses actions par coordination et négociation avec les autres agents. En construisant un plan pour atteindre un but, l'agent se donne les moyens d'y parvenir et donc s'engage à accomplir les actions qui satisfont ce but : L'agent croit qu'il est en mesure d'exécuter tout le plan qu'il a élaboré, ce qui le conduit (ainsi que les autres agents) à agir en conséquence [5][7].

- Adaptabilité : Un agent adaptatif est un agent capable de contrôler ses aptitudes (communicationnelles, comportementales, etc.) selon l'agent avec qui il interagit. Un agent adaptatif est un agent d'un haut niveau de flexibilité.

En conclusion, on appelle agent intelligent un agent qui comporte une ou plusieurs des caractéristiques présentées ci-dessus.

2.4 Communication dans un SMA

Un agent doit maintenir une représentation du monde et celle des autres agents formant le SMA pour raisonner pleinement ; le résultat sera d'autant meilleur si la représentation est complète. Pour construire cette représentation, l'agent doit acquérir des connaissances sur ses homologues et sur l'environnement en communiquant et en percevant [6]. Cette acquisition peut être directe en effectuant une requête explicite par un agent, afin de maintenir sa propre représentation des connaissances, ou indirecte, en inférant des connaissances sur les autres agents à partir des actes de communication qui ont été mis en œuvre pour arriver à un but commun.

2.4.1 Architecture de communication

La communication entre agents peut être organisée selon quatre schémas différents :

- Réseau en anneau [figure 2.3] : Les interactions dans ce genre d'organisation sont trop lentes. Tous les messages circulent dans l'anneau dans un même sens jusqu'à ce qu'ils se rendent à leurs destinations. Un message doit passer au plus par n-2 (n : nombre d'agents) agents avant de se rendre à destination.

- Réseau en bus [figure 2.4] : Dans cette architecture, les agents sont organisés en ligne. Les messages circulent dans les deux sens sur cette ligne. Un message doit passer au plus

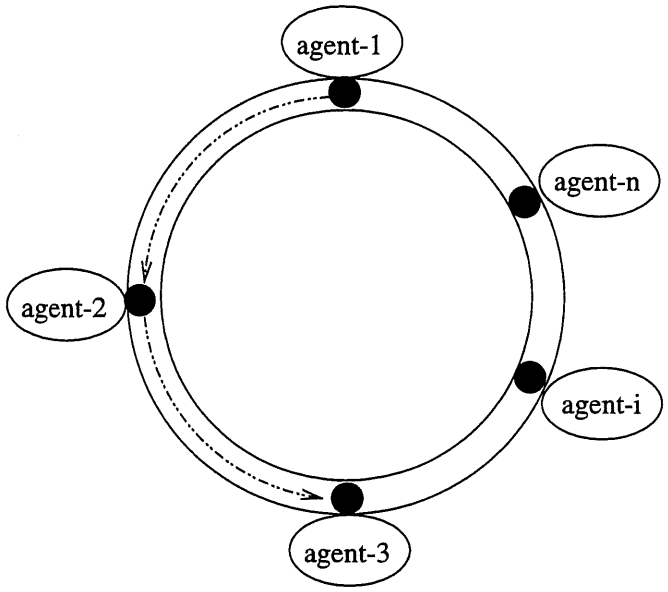

Figure 2.3 - *Réseau en anneau*

par (n-2)/2 agents (n : nombre d'agents) avant de se rendre à sa destination. Pour faire communiquer un message, un agent émet deux copies du même message. La première copie est émise dans un sens et l'autre dans le sens contraire.

- Réseaux en étoile [figure 2.5] : Ils présentent l'avantage de l'accès rapide entre le superviseur et les autres agents. La nature bidirectionnelle des connexions rend complexe la gestion des interactions inter-agents ;

- Réseaux hybrides [figure 2.6] : Ce type d'organisation est une solution hybride reliant deux types d'organisation : un réseau en bus et un réseau en étoile.

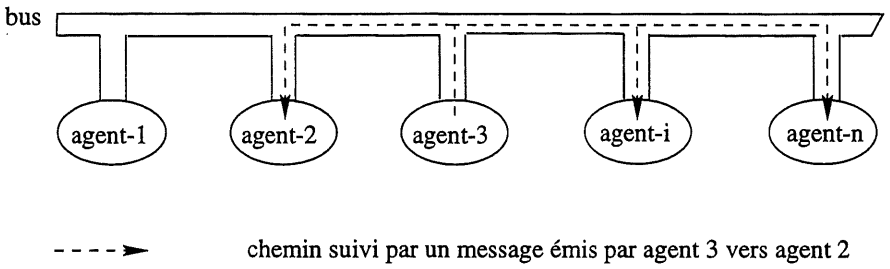

Figure 2.4 - *Réseau en bus*

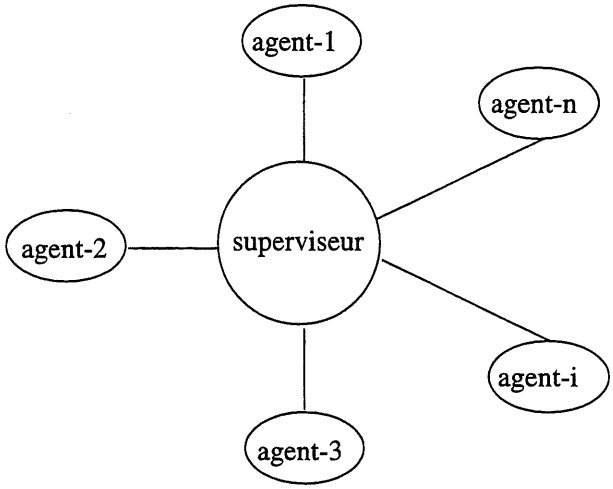

Figure 2.5 - *Réseau en étoile*

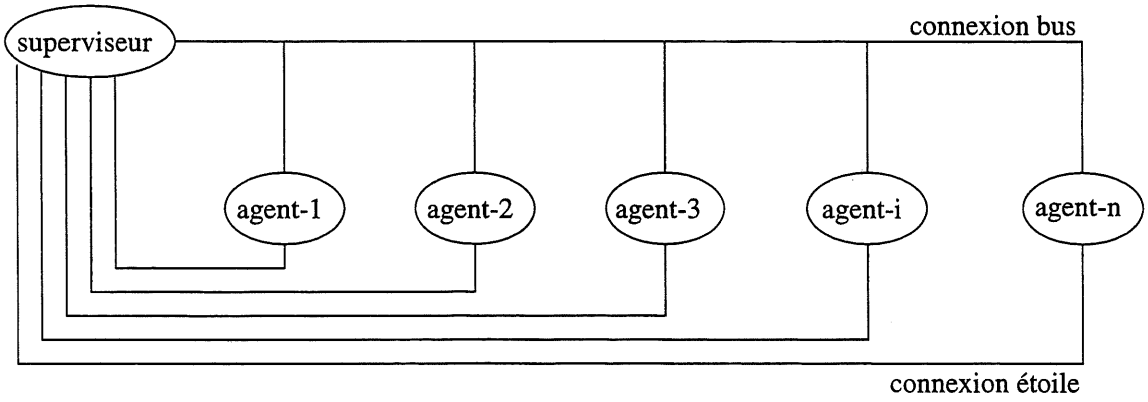

Figure 2.6 - *Réseau hybride*

2.4.2 Type de communication

Une communication dans les SMA n'est pas une simple tâche d'entrée-sortie, mais doit être modélisée comme un acte pouvant modifier l'état interne de l'agent récepteur. L'agent peut être amené à remettre en cause son plan d'action ou à modifier ses croyances.

Une communication peut correspondre à une information, une requête ou une interrogation ; elle doit avoir une sémantique reconnue par l'agent récepteur. La réception d'un message par un agent doit aider à une convergence vers la solution. Tout cela demande une bonne étude (par l'agent émetteur) du contenu du message lors de sa composition. Dans [10], il y a une distinction entre les différents types de communication :

- communication sélective ou diffusée : Par opposition à la diffusion, une communication sélective s'adresse à un nombre restreint d'agents, elle suppose donc un critère de sélection ;
- communication sollicitée ou non sollicitée : Une communication peut être demandée par un autre agent ;
- communication avec ou sans accusé de réception : Dans le cas d'une communication avec accusé de réception, l'émetteur attend une confirmation de la bonne réception du message.

2.4.3 Mode de communication

Il existe deux principaux modes de communication : la communication par envoi de messages et la communication par partage d'informations.

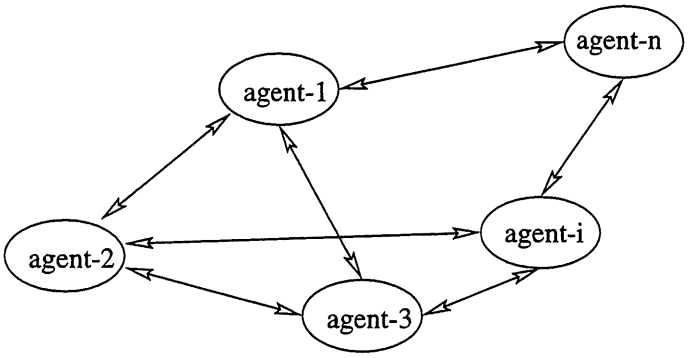

Figure 2.7 - *Communication par envoi de messages*

Communication par envoi de messages

Les agents sont en liaison directe et envoient leurs messages directement et explicitement au destinataire [figure 2.7]. La seule contrainte est la connaissance de l'agent destinataire : «Si un agent A connaît l'agent B, alors il peut entrer en communication avec lui» [47]. Les systèmes fondés sur la communication par envoi de messages relèvent d'une distribution totale à la fois de la connaissance, des résultats et des méthodes utilisées pour la résolution du problème.

Communication par partage d'informations

Les composants ne sont pas en liaison directe mais communiquent via une structure de données partagées, où l'on trouve les connaissances relatives à la résolution (état courant du problème) qui évoluent durant le processus d'exécution [figure 2.8]. Cette manière de communiquer est l'une des plus utilisées dans la conception des systèmes distribués. Le meilleur exemple d'utilisation de ce mode de communication est l'architecture de blackboard, on parle plutôt de sources de connaissances que d'agents. Ce mode de communication n'existe

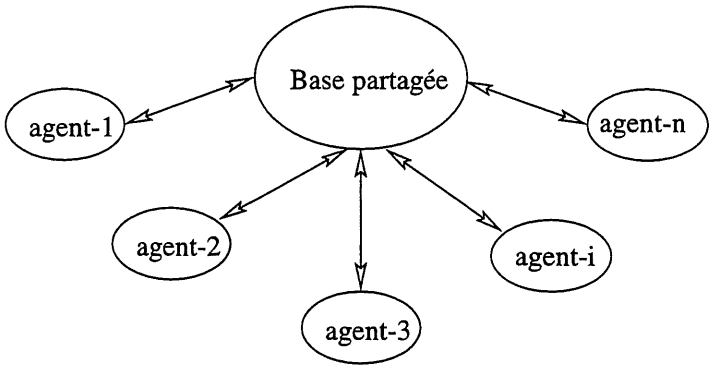

Figure 2.8 - *Communication par partage d'informations*

pas dans les SMA où l'on ne dispose que d'une vision partielle du système, alors que la communication par partage d'informations suppose l'existence d'une base partagée sur laquelle les composants viennent lire et écrire.

2.4.4 Protocole de communication

De nombreux protocoles de communication sont présentés dans la littérature des SMA. Par exemple, dans le système de Gaspar [23], un protocole simple exprimé par quatre types de message correspond aux modes de communication entre les agents.

Dans le système développé par Sian [43], le protocole met en œuvre une méthode d'apprentissage pour un SMA [figure 2.9]. À partir d'informations locales, chaque agent déduit de nouvelles hypothèses et maintient leur degré de confiance par interaction avec les autres agents via un tableau noir. Chaque agent comporte un module d'apprentissage, un module mémorisant l'expérience et un module de traitement utilisant cette expérience pour générer les actions.

Le protocole proposé dans ce système comporte huit primitives. Elles expriment les actes

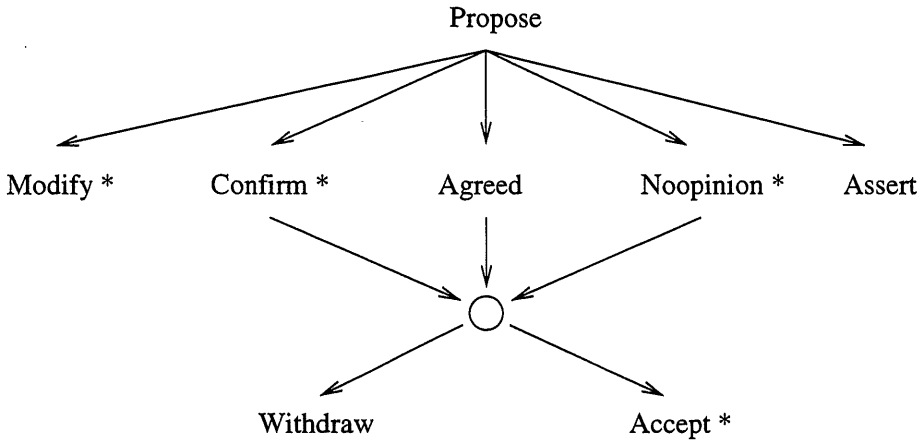

Figure 2.9 - *Protocole d'échange de connaissances proposé par Sian [43]*

de langage pour l'émission d'une hypothèse non modifiable (Assert), la proposition d'une nouvelle hypothèse (Propose), la modification d'une hypothèse émise précédemment (Modify), l'acceptation d'une hypothèse (Agree), le désaccord vis-à-vis d'une hypothèse de la validité d'une hypothèse (Disagree), l'absence d'opinion vis-à-vis d'une hypothèse (Noopinion), la confirmation de la validité d'une hypothèse (Confirm), le retrait d'une hypothèse précédemment proposé (Withdraw).

Le protocole de communication réduit les échanges possibles entre les agents en définissant les séquences possibles d'utilisation de ces primitives. Il consiste en trois phases : génération d'hypothèse, coopération afin de s'accorder sur les hypothèses et intégration des informations résultant des échanges.

Des protocoles plus complexes ont été élaborés par Kant et Woo [11] qui se sont basés sur les résultats d'études linguistiques concernant l'expression des posés. Les diverses intentions de communication sont exprimées au sein de ces protocoles.

Nous citons aussi le protocole KQML (Knowledge Query and Manipulation Language [21]) qui s'impose de plus en plus comme un standard de facto. En effet, le nombre des utilisateurs du protocole KQML n'a pas cessé d'augmenter depuis 1993.

KQML est en même temps un langage et un protocole par lequel des agents distants peuvent échanger des messages. Les enveloppes contenant les messages peuvent contenir n'importe quel genre de données. Le format de ces données n'est pas dicté par KQML. KQML fournit une grande variété de protocoles pour l'échange des messages simples, offrant la possibilité d'éditer et de souscrire des protocoles, de diffuser des messages, etc. Les agents peuvent envoyer des expressions à d'autres agents distants et recevoir (sur option) les réponses de celles-ci. D'un autre coté, un agent peut déclarer sa bonne volonté de recevoir des expressions provenant d'autres agents distants et de répondre (sur option) à celles-ci [21].

Comme nous le voyons, aucune normalisation de ces protocoles n'est actuellement disponible. Couvrant divers aspects des échanges qui peuvent se dérouler dans un SMA, le regroupement de ces protocoles en une structure unique est d'un immense intérêt.

2.5 Coopération entre agents

La coopération entre agents est un important concept vis-à-vis le fonctionnement d'un SMA. En effet, une résolution distribuée d'un problème est le résultat de l'interaction coopérative entre les différents agents du système. Dans le cadre d'une telle dynamique collective, un agent doit disposer en plus de la connaissance reflétant son degré d'implication dans cette dynamique (croyance, buts, intentions, engagements, modèle de soi et d'autrui) d'un certain

nombre de compétences nécessaires pour la coopération. Il doit pouvoir :

- agir dans un environnement ;
- communiquer directement ou indirectement avec d'autres agents ;
- percevoir partiellement son environnement ;
- mettre à jour le modèle du monde environnant ;
- intégrer des informations venant des autres agents ;
- interrompre ses tâches pour aider d'autres agents ;
- déléguer la tâche qu'il ne sait pas résoudre à un autre dont il connaît les compétences ;
- satisfaire ses objectifs en fonction des éléments précédents.

Ces caractéristiques forment les qualités essentielles d'un agent coopératif. Nous présentons dans ce qui suit, quatre approches pour la coopération entre agents : la négociation de contrats, l'échange de résultats intermédiaires, l'approche organisationnelle et la planification locale.

2.5.1 Coopération par négociation de contrat

La négociation est la technique la plus employée pour résoudre des vues incohérentes ou pour atteindre un accord sur la façon de travailler ensemble. Dans les approches fondées sur le principe de la négociation, des alliances temporelles sont définies entre agents dans un réseau. Deux formes d'allocation de sous-tâches fondées sur la négociation sont : le «Contrat

Net Protocol» [15] [14] et la négociation multiniveaux [42] qui implique plusieurs itérations du processus de négociation. Dans ces approches, les rôles et les tâches sont alloués de manière dynamique en fonction de la disponibilité des ressources et des capacités effectives des agents.

Du fait du manque de vue globale de l'état du réseau à un instant donné, des tâches similaires risquent également d'être exécutées dans deux contrats différents. Enfin, un risque de récursivité peut apparaître, du fait que les agents peuvent assurer des rôles différents pour différents contrats.

2.5.2 Coopération par échange de résultats intermédiaires

Ici, l'objectif est de traiter l'existence de vues inconsistantes. Dans des systèmes à base de connaissances importantes, il est en effet improbable qu'une base de données totalement consistante puisse être maintenue. L'existence d'inconsistances est ainsi inévitable et peut même permettre de considérer des interprétations différentes. Les inconsistances sont alors résolues en échangeant les résultats de haut niveau issus de l'interprétation de données de bas niveau. Cette résolution des ambiguïtés forme une partie intégrale de la tâche de résolution de problème. Permettre un libre-échange d'informations pour résoudre les ambiguïtés conduit rapidement à des coûts de communication excessifs [18]. Dans une étude de la métaphore de la communauté scientifique, Kornfeld [30] a montré comment réduire la communication de solutions partielles à un petit nombre d'agents.

2.5.3 Coopération par approche organisationnelle

L'approche organisationnelle constitue un compromis entre la négociation de contrat et le partage de résultats intermédiaires en définissant des alliances parmi les agents appartenant à la même organisation : les agents possèdent ainsi une vue de haut niveau de leur rôle dans le processus de résolution et de leurs relations avec les autres. Ces rôles et relations sont préassignés d'une façon plutôt permanente, chaque fois qu'une nouvelle organisation est créée pour exécuter une tâche.

Des stratégies de contrôle hiérarchiques peuvent être développées comme dans le système IBIS [48], qui est une adaptation du concept de tableau noir. Dans ce système, les sources de connaissances peuvent communiquer à travers le tableau noir en utilisant un schéma hiérarchique des connexions existantes entre elles. Seuls des messages de contrôle peuvent être échangés entre les sources de connaissances ; les données doivent toujours être partagées à travers la base de données.

Le problème principal des agents travaillant en coopération est que des situations conflictuelles surviennent fréquemment à cause d'environnements imprévisibles et changeants constamment. La capacité à reformuler les buts et à réallouer les tâches pour résoudre les inconsistances qui peuvent survenir dans une interprétation est essentielle. C'est pourquoi certaines approches permettent de modifier dynamiquement la structure organisationnelle. Par exemple, Corkill et Lesser utilisent des structures organisationnelles pour réduire les échanges d'informations entre les agents [34]. Ces structures sont élaborées de manière dynamique et concurrente par les agents. Durfee et Lesser [35] suggèrent qu'un métaniveau de contrôle est nécessaire pour guider la génération de nouvelles structures et pour identifier les échecs des structures existantes.

2.5.4 Coopération par planification locale

La planification dans les systèmes d'IA classique repose sur l'hypothèse d'un univers statique (seules les actions du planificateur sont considérées) ; cet aspect est incompatible avec l'approche multiagents [20]. Les SMA, en revanche, offrent des possibilités de négociation autorisant une gestion locale des conflits et une planification dynamique. En effet, du fait de l'intervention d'autres agents, un plan peut être remis en cause. L'agent doit, pour cela, alterner entre planification et exécution et réviser des parties de son plan. La planification dans les SMA est une planification distribuée : il n'existe pas de plan global et chaque agent construit son propre plan en coordination avec les autres (cas d'agents coopératifs). Certains SMA présentent une planification centralisée ; un organe central se chargera de la gestion des conflits et de l'élaboration d'un plan global.

2.6 Résolution de conflit

Les agents coopératifs ont besoin d'éviter autant que possible les situations conflictuelles pour résoudre un problème. Pour ce faire, ils peuvent être amenés à coordonner leurs activités et négocier leurs actions pour arriver à une solution.

2.6.1 Coordination

Les agents travaillent sur des problèmes dont les solutions sont utiles pour les autres agents. Leur travail doit être temporellement coordonné. La coordination [17] permet aux agents de considérer toutes les tâches (aucune tâche n'est ignorée) et de ne pas dupliquer le

travail. La coordination des actions est liée à la planification et à la résolution des conflits, car c'est à ce niveau qu'on tient compte des actions (plans) des différents agents.

Dans les systèmes d'IAD, la coordination des actions des agents peut s'organiser suivant deux schémas principaux [19] : une coordination au moyen d'un système capable de déterminer et de planifier (globalement) les actions des différents agents ou, à l'inverse, on décide de donner une autonomie totale aux agents qui, à leur tour, identifient les conflits pour les résoudre localement.

On peut distinguer deux types de coordination : la coordination causée par la gêne (problème de navigation : les agents doivent coordonner leurs plans de navigation pour s'éviter mutuellement) et la coordination causée par l'aide (la manutention : dans un environnement multirobots, les agents doivent synchroniser leurs actions pour pouvoir agir efficacement et transporter un objet [39]).

2.6.2 Négociation

Les activités des agents dans un système distribué sont souvent interdépendantes et entraînent des conflits. Pour les résoudre, il faut considérer les points de vue des agents, les négocier et utiliser des mécanismes de décision concernant les buts sur lesquels le système doit se focaliser [12]. La négociation est caractérisée par :

- un nombre faible d'agents impliqués dans le processus ;
- un protocole minimal d'actions : proposer, évaluer, modifier et accepter ou refuser une solution.

Le processus de négociation ne consiste pas forcément à trouver un compromis, mais il peut s'étendre à la modification des croyances d'autres agents pour faire prévaloir un point de vue. Pour mener à bien le processus de négociation [14] [13], il est nécessaire de suivre un protocole qui facilite la convergence vers une solution. Voici un exemple de structure de négociation entre deux agents A et B :

- A fait une proposition ;
- B évalue cette proposition et détermine la satisfaction qui en résulte ;
- Si B est satisfait, alors arrêt, sinon B élabore une contre-proposition et donne des arguments ;
- Si A considère les arguments de B, alors reprendre ce schéma en interchangeant A et B, sinon faire intervenir un troisième agent suivant le principe de la dynamique des échanges proposé dans [12].

La dynamique des échanges effectuée par un agent consiste à ordonner les agents intervenants et à tenir compte des différents points de vue, de manière à ce que le groupe aboutisse à une décision.

2.7 Interaction entre agents

L'interaction est une mise en relation dynamique de deux ou plusieurs agents par le biais d'un ensemble d'actions réciproques. Les interactions s'expriment ainsi à partir d'une série d'actions dont les conséquences exercent en retour une influence sur le comportement futur

des agents. Les agents interagissent le long d'une suite d'événements pendant lesquels ils sont d'une certaine manière en contact les uns avec les autres, que ce contact soit direct ou qu'il s'effectue par l'intermédiaire d'un autre agent ou de l'environnement.

Les interactions sont non seulement la conséquence d'actions effectuées par plusieurs agents en même temps, mais aussi l'élément nécessaire à la constitution d'organisations sociales. C'est par les échanges qu'ils entretiennent, par les engagements qui les lient, par l'influence qu'ils exercent les uns sur les autres que les agents sont des entités sociales. Les groupes sont donc à la fois les résultats d'interactions et les lieux privilégiés dans lesquels s'accomplissent les interactions. C'est pourquoi il est généralement impossible d'analyser des organisations sociales sans tenir compte des interactions entre leurs membres. L'étude des méthodes d'interaction entre agents consiste à enrichir le comportement de ces derniers et en intégration, de manière simple, de nouveaux agents.

2.8 Outils de développement de SMA

2.8.1 Introduction

Nous remarquons ces dernières années l'évolution constante des applications dans tous les domaines (multimédia, base de données, communication, simulation, contrôle, etc.). Cependant, la complexité de ces applications n'a pas cessé d'augmenter. Et si les versions des logiciels d'aujourd'hui changent plus fréquemment qu'avant, c'est grâce à l'évolution des outils de programmation de haut niveau qui tendent de plus en plus à simplifier les tâches de programmation, de test et de maintenance. Ceci réduit énormément les délais de mise en marché des applications en entraînant ainsi une baisse des prix de celles-ci. Le domaine des

SMA n'a pas encore eu sa part complète en ce qui concerne les outils de développement. En effet, les outils multiagents existants ne répondent pas à tous les besoins des programmeurs. Ces outils présentent en général des avantages et des inconvénients. Ceci implique l'absence sur le marché d'un outil complet pour le développement de SMA. Nous présenterons dans ce qui suit quelques outils multiagents suivis par une évaluation comparative.

2.8.2 Langage Agent Logiciel Objet (LALO)

Le centre de recherche en informatique à Montréal (CRIM) a développé au cours des trois dernières années l'outil LALO (Langage Agent Logiciel Objet) permettant la création de SMA [24]. LALO est un langage permettant de définir le comportement d'un agent en utilisant les concepts de croyances, de capacités, de décisions et d'engagements. Les éléments de base du langage sont les croyances et les tâches qui représentent deux concepts intimement liés au temps. Un temps est associé à chaque croyance et à chaque tâche :

```
<tâche>::= DO <action> |AT <expression temporelle> DO <action>
<croyance> ::= <fait> | NOT <fait> | AT <expression temporelle> <fait>
               | AT <expression temporelle> NOT <fait>
```

Trois actions de communication sont prédéfinies dans LALO. La première action, `INFORM`, permet aux agents d'échanger de l'information. Sa forme est :

```
<informer> ::= INFORM{wich: <agent> ; of: <croyance>}
```

Dans cette forme, `<agent>` est le nom de l'agent auquel est destiné le message et `<croyance>` le fait que l'expéditeur croit vrai. La deuxiemme action, `REQUEST`, permet à un agent de

requérir les services d'un autre agent en lui demandant d'effectuer une tâche. Sa forme est :

```
<requérir> ::= REQUEST{to: <agent> ; task: <tâche>}
```

La dernière action de communication permet à un agent d'annuler une requête formulée précédemment. Sa forme est :

```
<annuler> ::= UNREQUEST{to: <agent> ; task: <tâche>}
```

Le protocole de communication utilisé est KQML [21] permettant ainsi aux agents LALO de communiquer avec des agents développés avec d'autres outils. En plus de KQML, les agents LALO peuvent également traiter des messages HTTP à titre de serveur ou de client.

Les deux éléments essentiels de l'environnement LALO sont : une librairie C++, `libAgent` et un compilateur. La librairie `libAgent` contient les classes d'agents prédéfinies dans le système. Cette librairie contient, entre autres, la classe `BasicAgent` qui implante les mécanismes de communication de base et la boucle de base de fonctionnement de l'agent. L'utilisateur peut augmenter le nombre de classes d'agents disponibles en créant sa propre librairie. Cependant, toute classe d'agents créée par l'utilisateur doit hériter de l'une des classes de la librairie `libAgent`.

Contrairement à la plupart des environnements de développement multiagents actuellement disponibles, les agents LALO sont compilés plutôt qu'interprétés. En effet, un compilateur traduit l'agent décrit en LALO en code source C++ qui peut par la suite être compilé avec un compilateur C++ compatible.

2.8.3 Java Agent Template (JAT)

Le JAT [22] fournit un descripteur entièrement fonctionnel, écrit entièrement en langage Java, pour construire des agents communiquant avec d'autres agents distants. Bien que les parties du code qui définissent chaque agent soient portables, les agents de JAT ne sont pas migrateurs, mais ils ont plutôt une existence statique sur un simple serveur. Ce comportement est contraire à beaucoup d'autres nouvelles technologies d'agent. Cependant, en utilisant le Java RMI (Remote Method Invocation) [45], des agents de JAT pourraient dynamiquement passer à un autre serveur par l'intermédiaire d'un d'agent résidant sur ce serveur. Actuellement, tous les messages d'agent utilisent le KQML comme protocole de messagerie. Le JAT inclut la fonctionnalité pour échanger dynamiquement des ressources qui peuvent inclure des classes de Java, des fichiers de données et l'information relative aux messages du KQML.

On peut exécuter des agents de JAT en tant qu'applications autonomes ou en tant qu'applets en utilisant l'Appletviewer. L'architecture du JAT a été particulièrement conçue pour tenir compte de la spécialisation des principaux composants fonctionnels comprenant la transmission de messages à un bas niveau, l'interprétation de messages et la manipulation de ressources. En conséquence, le JAT devrait être utilisé comme une plate-forme pour établir un éventail d'agents pour différents domaines d'applications.

2.8.4 Actalk

Actalk est une plate-forme qui implante différents types de modèles d'objets actifs (appelés aussi acteurs) [49]. Il est fondé sur l'existence d'un noyau qui modélise des objets actifs communiquants par envoi de messages asynchrones. La communication asynchrone, principe de base des langages d'objets actifs ; est implantée par une communication synchrone puis

par dépôt du message dans un tampon (boîte aux lettres) ; il y a ainsi dissociation entre envoi et interprétation du message [28]. Les principales classes qui décrivent les trois composantes de base d'un objet actif sont les suivantes :

- La classe `Adresse` : représente l'adresse d'un objet actif. Elle définit la politique de réception de toutes les requêtes faites à l'objet actif. Elle décrit donc les différents types de communication.
- La classe `Activity` : représente l'activité interne d'un objet actif. Elle décrit la manière dont les messages sont sélectionnés, séquencés et activés en contrôlant les messages placés dans la boîte aux lettres. Elle utilise un processus pour retirer continuellement les messages présents dans la boîte aux lettres et lancer leur interprétation spécifiée par le comportement de l'objet actif.
- La classe `ActiveObjet` : décrit le comportement de l'objet actif et la réponse aux requêtes qui lui sont transmises par l'objet activité.

Ces trois classes de base sont ensuite progressivement spécialisées (sous-classées) pour définir différents modèles. Actalk fournit à Smaltalk quelques caractéristiques des systèmes ouverts [1] telles que les multiples activités et les communications simultanées. Une des originalités de la dernière version d'Actalk est la réification de l'activité d'un objet actif. Cette nouveauté, quoique puissante, ne peut toutefois rendre réellement autonome un objet actif. L'activité de ce dernier dépend des autres objets actifs, il reste inactif s'il ne reçoit pas de message. Pour améliorer son autonomie, différentes plates-formes proposent d'enrichir cet objet actif d'une fonction lui permettant de contrôler le traitement des messages reçus en considérant son état interne [9] [36].

2.8.5 Voyager

`Voyager` a été conçu par ObjectSpace [38] pour aider les développeurs à produire rapidement des systèmes distribués. `Voyager` est entièrement développé en Java, utilisant le model d'objet du langage Java. Il permet d'utiliser des syntaxes de messages simples et régulières pour construire des objets distants, leurs envoyer des messages, les faire migrer d'un programme à un autre. Il combine la puissance des agents mobiles et de l'invocation des méthodes à distance.

Beaucoup d'applications réparties ou multiagents peuvent tirer bénéfice des objets actifs et mobiles. Voyager permet à des agents, objets autonomes, de se déplacer et de continuer d'exécuter pendant qu'ils se déplacent. De cette façon, les agents peuvent agir indépendamment au nom d'un client, même si le client est débranché ou indisponible.

2.8.6 Analyse

Après avoir survolé quelques outils de développement de SMA, nous pouvons dresser une liste regroupant les principaux mécanismes intégrés dans ces outils. Ces mécanismes peuvent être scindés en deux groupes :

- L'ensemble des mécanismes d'aide à la création des agents intelligents regroupant :
  - les formalismes pour représenter les croyances d'agent;
  - formalismes pour représenter les différents comportements d'agent.

caractéristiques	LALO	Voyager	Actalk	JAT
croyances	représenté	non représenté	représenté	non représenté
comportement	représenté	non représenté	représenté	non représenté
mobilité	fixe	mobile	fixe	fixe
multitâches	oui	oui	oui	oui
communication	pauvre	riche	pauvre	riche
langage de programmation	C++	java	Smaltalk	java

TABLEAU 2.1 - *Tableau comparatif des quatre outils multiagents*

- L'ensemble des mécanismes d'aide à la création des agents communicants regroupant :
  - les fonctions de base pour permettre à un agent de communiquer et d'interagir avec son monde externe;
  - les protocoles de communication pour structurer les échanges de données entre agents;
  - la mobilité d'agent lui permettant de migrer d'un système à un autre sans causer des perturbations sur le comportement des autres agents.

Aucun, parmi les outils que nous avons présentés, n'inclut tous ces mécanismes. Chacun de ces outils présente des avantages et des inconvénients par rapport aux autres. Le tableau 2.1 présente une étude comparative non exhaustive entre ces différents outils. Les critères de comparaison sont basés sur les principaux concepts évoqués par les outils de développement multiagents existants.

En effet, le nombre de fonctions de communication offert par l'outil LALO pour un agent se trouve insuffisant en comparaison avec l'outil `Voyager`. Par contre, il offre des méthodes

intéressantes pour la représentation des connaissances et des comportements d'agent. À l'opposé de LALO et Actalk, l'outil JAT ainsi que `Voyager` sont riches en mécanismes de communication entre agents. Ces deux outils sont conseillés pour le développement d'agents communicants. Malheureusement, l'absence de formalismes de représentation de connaissances et du comportement d'agent représente le point faible de ces derniers.

Le projet de recherche EGSMA dont les travaux se déroulent au Département de génie électrique et de génie informatique de l'Université de Sherbrooke vise à remédier aux inconvénients des outils existants. Les objectifs de ce projet consistent à la conception et à la réalisation d'un outil complet et général pour le développement de SMA. La complétude et la généralité de l'outil signifient que celui-ci offre aux programmeurs multiagents une librairie de fonctions riche en mécanismes de représentation de :

- agents intelligents (les connaissances, le comportement, les tâches, le contrôle, etc.);
- agents communicants (échange de messages, protocole de communication, interaction, diffusion de message, synchronisation, migration, etc.).

L'ensemble des fonctionnalités offertes par l'outil EGSMA doit être complet et simple à utiliser pour simplifier et accélérer le développement des SMA. Nous présenterons dans le chapitre suivant notre participation pour la mise en œuvre de cet outil.

Chapitre 3

Un outil de communication pour les SMA

3.1 Introduction

Nous proposons dans ce chapitre un outil de communication entre agents pour les SMA. Le développement de cet outil est une contribution à la réalisation du projet EGSMA. Les recherches bibliographiques effectuées sur les SMA dans le chapitre 2 nous ont guidés pour définir les spécifications de l'outil OCA.

3.2 Interface de communication pour un agent

Nous avons présenté dans la figure 2.1 une structure générale d'un agent qui représente une synthèse des différentes structures présentées dans la littérature. Nous rappelons les principales composantes de cette structure :

- le savoir-faire ;
- les croyances ;
- le contrôle ;
- l'expertise ;
- la communication.

Le savoir-faire, les croyances, le contrôle et l'expertise représentent l'intelligence d'un agent. La communication représente une interface d'interaction à travers laquelle l'agent peut échanger des informations avec son monde externe au sein d'un SMA.

L'objectif de l'outil EGSMA est de faciliter la tâche des programmeurs d'applications multiagents pour le développement d'agents intelligents et communicants en lui offrant un modèle d'agent basé sur la structure d'agent proposée dans la figure 2.1.

Notre contribution dans ce projet consiste à la conception et à la réalisation de la partie interface de communication. Cette interface offrira à un agent les fonctions de base de communication pour toute interaction, sans difficulté, avec son monde externe.

3.3 Besoins de communication pour un agent

Pour déterminer l'ensemble des fonctionnalités de l'interface de communication d'un agent, nous allons survoler brièvement les différents concepts de communication évoqués dans le chapitre 2. Ces différents concepts nous guideront à déterminer les différents besoins de communication pour un agent dans un SMA.

3.3.1 Architecture de communication

Les trois principales architectures de communication qu'on trouve dans la littérature sont les suivantes :

- Communication en anneaux : Dans cette architecture, les agents sont organisés sous forme d'anneau. Chaque agent ne communique qu'avec les deux agents adjacents. Durant une communication entre deux agents, l'information circule dans un seul sens. Un message émis par un agent vers une destination transitera d'un agent à un autre jusqu'à ce qu'il se rende à sa destination. Un agent peut recevoir alors un message qui ne lui est pas destiné. Dans ce cas, ce dernier renvoie le message à l'agent suivant. Le même processus se reproduira pour les agents suivants jusqu'à ce que le message parvienne à sa destination.

- Communication en bus : dans cet architecture, les agents sont organisés en ligne. À la différence d'une architecture en anneau, les messages circulent entres deux agents dans les deux sens. Les messages parviennent à leurs destinations en transitant par des agents intermédiaires.

- Communication hybride : cette architecture réunit les deux premières architectures.

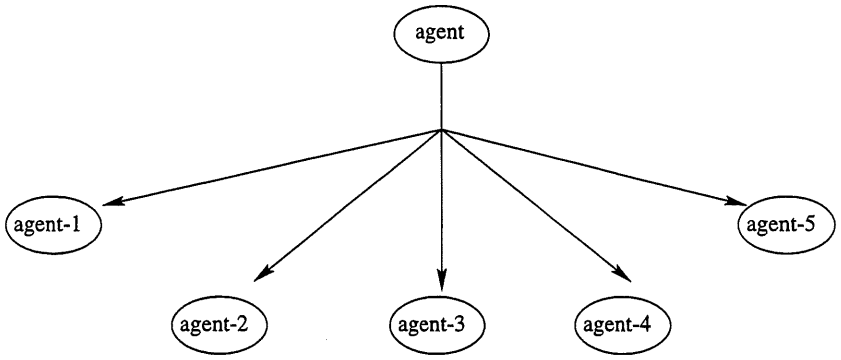

Figure 3.1 - *La diffusion d'un message*

- communication en étoile : Dans ce genre de communication, les agents communiquent entre eux indirectement via un agent intermédiaire. Ce dernier est en communication directe avec tous les agents. L'agent intermédiaire ou l'agent superviseur reçoit alors des messages qui ne lui sont pas destinés et les achemine à leurs destinations.

Ce que nous pouvons retenir de ces quatre types de communication pour les spécifications de l'interface de communication est que : deux agents peuvent avoir une communication directe ou une communication indirecte via un agent intermédiaire.

3.3.2 Type de communication

Les principaux types de communication qu'un agent peut effectuer sont :

- Communication diffusée : Un agent peut diffuser un message à tous les agents en communication avec lui [figure 3.1].
- Communication sélective : Pour envoyer un message, un agent doit sélectionner une ou plusieurs destinations.

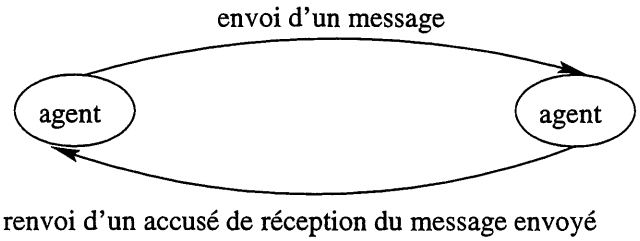

Figure 3.2 - *Communication avec accusé de réception*

- Communication avec un accusé de réception : Pour s'assurer de la bonne réception d'un message, un agent peut exiger un accusé de réception de l'agent destinataire. Ce dernier doit alors renvoyer un message à l'expéditeur pour confirmer la réception d'un message [figure 3.2].

- Communication sollicitée : Un agent demande une autorisation de dialogue à son homologue avant de lui émettre des messages. Ce dernier, suivant son état (libre ou occupé), peut accepter (avec ou sans restriction) cette demande ou la refuser complètement [figure 3.3].

- Communication non sollicitée : Les agents communiquent entre eux directement sans passer par un préavis.

Tous les types de communication cités ci-dessus correspondent à des fonctions à inclure dans l'interface de communication d'un agent.

3.3.3 Mode de communication

Dans la littérature, on distingue deux modes de communication. Le premier est basé sur le partage d'information et le deuxième sur l'envoi et la réception de messages. Le premier mode

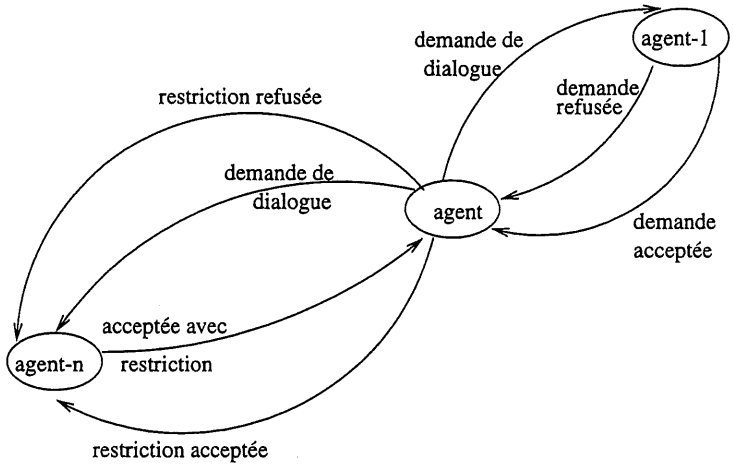

Figure 3.3 - *Différents cas de sollicitation de dialogue entre deux agents*

exige une mémoire partageable par l'ensemble des agents formant le SMA. Nous pensons que ce mode de communication peut être implanté même si cette contrainte n'est pas vérifiée. En effet, l'information à partager peut être déclarée sur un agent dédié. Ce dernier reçoit des requêtes des autres agents (par envoi et réception de messages) pour consulter ou mettre à jour l'information partagée. C'est pourquoi alors nous traitons seulement la communication par envoi de message dans ce travail.

3.3.4 Protocole de communication

Les protocoles de communication dans les SMA représentent des protocoles de haut niveau qui consistent à structurer et à réduire les échanges de messages entres agents. Plusieurs protocoles ont été élaborés dans la littérature. Aucune normalisation pour ces protocoles n'est disponible. Les protocoles de communication ne seront pas traités dans ce projet pour la simple raison que ces derniers dépendent du comportement et de l'organisation des agents au sein d'un SMA. Nous avons jugé qu'il sera plus intéressant d'inclure les protocoles de

communication dans l'outil OCA après avoir défini les formalismes de représentation de croyance et de comportement d'agent dans l'outil EGSMA.

3.3.5 Autres concepts à tenir en compte

Traitement et gestion des messages reçus par un agent : Un agent doit traiter tous les messages qu'il reçoit de son monde externe. Le traitement de ces messages peut se faire dès leur réception ou à un temps ultérieur. Dans le cas d'un traitement ultérieur, les messages reçus par un agent seront temporisés dans une boîte qui contiendra les messages en attente de traitement. À tout instant, l'agent peut récupérer de cette boîte les nouveaux messages un après l'autre pour leurs interprétations ou leurs traitements. Ainsi, l'agent peut se servir de cette boîte pour garder une copie des messages reçus.

Synchronisation : La poursuite d'un traitement propre à un agent nécessite parfois une information ou la collaboration d'un autre agent. L'agent émet alors un message sous forme de requête à son homologue puis attendra la réponse de ce dernier. L'attente d'un message par un agent signifie la suspension temporaire de ces traitements jusqu'à la réception du message attendu.

Rupture et suspention de dialogue [figure 3.4]: À tout moment et pour des raisons quelconques, un agent peut rompre ou suspendre un dialogue avec un autre agent. Pour la rupture d'un dialogue, il y a plusieurs manières de le faire. Avant de rompre un dialogue, un agent «poli» demande l'avis de son homologue. Ce dernier, suivant ses besoins, peut accepter la demande comme il peut la rejeter. Par contre, un agent «impoli» rompt un dialogue sans demander l'avis de son homologue. Au cours d'un dialogue, l'état d'un agent peut changer. À

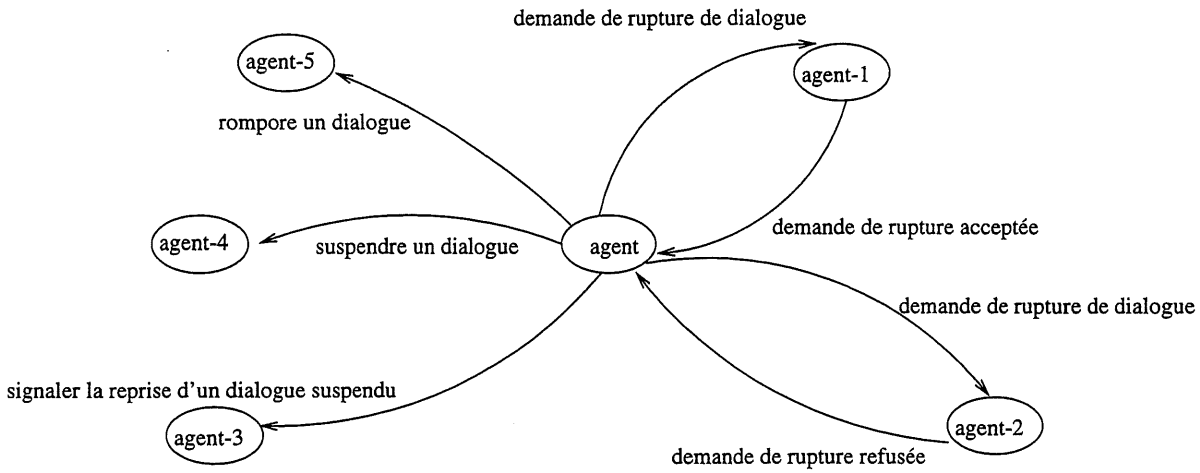

Figure 3.4 - *La rupture et la suspension de dialogue entre deux agents*

la suite d'un changement d'état (par exemple de libre à la saturation), un agent peut rompre le dialogue avec son homologue ou le suspendre temporairement et le reprendre dès qu'il se libère.

Localité d'un agent : Chaque agent est localisé par une adresse qui correspond à l'adresse IP de la machine dans laquelle il loge. Un agent mobile peut changer de localité durant son traitement. En cas de changement d'adresse, un agent peut informer ses homologues de sa nouvelle adresse. Ces derniers confirment à leur tour la réception de cet avis. Avant de changer physiquement son adresse, un agent doit s'assurer alors que son avis a été bien reçu par tous ses homologues. Le changement d'adresse se fait alors après la réception de la dernière confirmation. Un autre cas peut se présenter: l'agent en question peut changer son adresse physiquement en ignorant quelques-unes ou toutes les confirmations, suivant l'importance des dialogues qu'il entretient.

Agent client-serveur : On peut distinguer deux types d'agents :

- Agent serveur : Un agent serveur peut être sollicité par d'autres agents pour demander un service. Pour ce faire, ce dernier doit réserver un port de communication sur la machine sur laquelle il s'exécute pour répondre aux demandes de dialogue des agents clients.
- Agent client : À l'opposé d'un agent serveur, un agent client ne peut être sollicité pour un dialogue. Par contre, ce dernier peut solliciter un agent serveur pour un dialogue. Tous les agents sont des agents clients par défaut.

Traitement multitâche : En plus des actes de communication, un agent s'occupe de ses propres tâches relatives à l'application multiagents. Ces tâches ne doivent pas être bloquées par les tâches de communication. Un agent peut gérer plusieurs dialogues simultanément. Donc, il peut recevoir plusieurs messages de différentes provenances et, en même temps, émettre des messages à des différentes destinations.

Structure des messages : En plus de l'information utile à émettre, nous jugeons utile d'associer une priorité et un type à chaque message émis. Les messages reçus par un agent seront traités suivant leur ordre de priorité. Le type de message permet à un agent de classer les messages reçus en catégories.

Environnement hétérogène : Nous supposons qu'un SMA peut être implémenté sur un environnement hétérogène. Cet environnement peut être formé par un ensemble de machines distinctes, distantes et liées par un réseau de communication. Le système d'exploitation d'une machine peut différer d'une machine à une autre.

3.4 Conclusion

Les différents concepts de communication dans une société d'agents évoqués dans ce chapitre correspondront aux principales fonctionnalités de l'interface de communication d'un agent dans l'outil OCA. Ces différentes fonctions peuvent être classées en quatre catégories :

- Les fonctions de gestion de dialogues : la sollicitation, la rupture et la suspension de dialogue. Aussi, nous pourrons inclure la fonction de changement d'adresse dans cette catégorie.
- Les fonctions d'échange de messages : l'envoi, la diffusion et la réception de messages, etc.
- Les fonctions de gestion de messages : la temporisation des messages reçus, la classification de ces messages suivant un ordre de priorité, la signalisation de réception d'un message attendu.
- Les fonction de traitement de message : la récupération et la manipulation des messages temporisés et le traitement des messages dès leur réception.

Nous proposerons dans le chapitre suivant la conception générale de l'outil OCA basée sur la classification présentée ci-dessus et la mise en œuvre de celle-ci.

Chapitre 4

Conception et mise en œuvre

4.1 Introduction

L'outil EGSMA est conçu pour offrir aux utilisateurs un modèle d'agent intelligent et communicant. Ce modèle inclut les formalismes de représentation de croyances, les formalismes d'expression de comportements et une interface de communication pour toute interaction avec le monde externe d'un agent. Nous considérons ce modèle d'agent dans ce chapitre comme étant un modèle de classe nommé `agentCommunicant`. Pour la mise en œuvre d'un agent intelligent et communicant, l'utilisateur doit créer une classe dérivée de la classe `agentCommunicant` en lui rajoutant d'autres fonctions spécifiques aux besoins de l'application multiagents. L'outil OCA offre à l'utilisateur une première esquice du modèle d'agent communicant intégrant seulement les fonctions relatives aux tâches de communication. Nous proposons dans ce qui suit la structure générale de l'outil OCA.

4.2 Structure générale de OCA

En se basant sur la classification des fonctions de communication présentée dans le chapitre précédent, nous avons structuré l'outil OCA de la manière suivante [figure 4.1] :

- un contrôleur pour le traitement des demandes de dialogue (TDD) ;
- un contrôleur pour le contrôle d'émission et de réception de messages (CERM) ;
- une boîte de messages (BM);
- un contrôleur de traitement des événements externes (CTE) ;
- un contrôleur pour la communication avec les processus TDD, CERM et BM (CCP);
- et une zone pour la représentation des croyances et du comportement d'agent (ZRCC).

Les contrôleurs présentés ci-dessus sont considérés comme étant des processus indépendants les uns des autres. Chacun est dédié pour effectuer une tâche spécifique. Ces derniers s'exécutent en concurrence et communiquent entre eux par envoi de messages via l'invocation de méthodes. Nous avons opté pour une structure multi-processus pour permettre le traitement multi-tâches. Ceci nous permet d'éviter l'interblocage des différentes tâches de communication au sein d'un agent communicant.

Pour un utilisateur, la classe `agentCommunicant` est constituée essentiellement par le contrôleur de communication CCP, le contrôleur de traitement des événements CTE et la zone de représentation ZRCC.

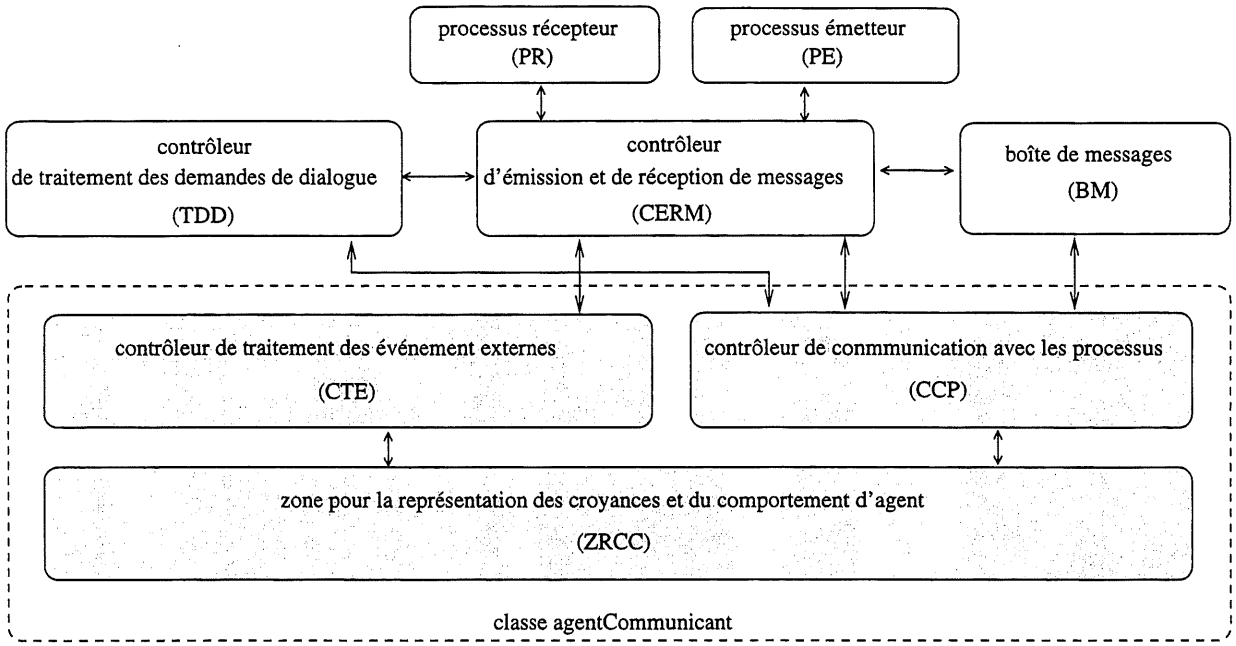

Figure 4.1 - *Structure générale de OCA*

4.2.1 Le contrôleur TDD

Ce processus n'est activé que par un agent serveur. Pour qu'un agent client puisse demander le dialogue à un agent serveur, il doit connaître l'adresse électronique et le port de ce dernier. Pour cela, tout agent serveur doit avoir un port de communication fixe sur la machine sur laquelle il s'exécute. Cette machine est identifiée par une adresse IP unique (par exemple: le port 25 sur la machine d'adresse IP pollux.gel.usherb.ca). Le rôle principal du contrôleur TDD est de:

- Réserver un port de communication: Sur une machine, il peut y avoir plusieurs ports de communication relatifs à des différentes applications. Comme, par exemple, le port de communication du ftp (protocole de transfert de fichiers) est le port fixe numéro 21. Donc, pour réserver un port de communication, il faut choisir un numéro de port non

utilisé par d'autres applications.

- Détecter toutes les connexions externes sur le port de communication. Une connexion externe correspond au fait à une demande de dialogue par un agent client. Un canal de communication sera établi entre l'agent serveur et l'agent client à la suite de toute connexion externe. Une copie du descripteur du canal établi sera passée au contrôleur CERM pour surveiller continuellement l'arrivée des messages envoyés sur celui-ci. Dans le cas d'une communication non sollicitée, la connexion externe sera acceptée automatiquement. Dans le cas contraire (une communication sollicitée), plusieurs cas peuvent se présenter :
  - L'agent serveur accepte le dialogue. Ce dernier doit alors émettre un message d'acceptation de dialogue à l'agent client.
  - L'agent serveur accepte de servir l'agent client avec des restrictions. Ces restrictions sont de type temporelle. L'agent client reçoit la restriction temporelle du serveur qui n'est qu'une date à partir de laquelle l'échange de messages pourrait débuter. Si cette condition est acceptée, un message d'acceptation de condition sera envoyé au serveur et le canal établi entre les deux agents restera ouvert. Dans le cas contraire, un message de refus sera renvoyé à l'agent serveur et le canal sera fermé.
  - L'agent serveur rejette la demande de service de l'agent client. Un message de refus sera alors renvoyé à l'agent client et le canal établi sera fermé.

Le contrôleur TDD est caractérisé par des paramètres qui définissent les intentions d'un agent vis-à-vis l'acceptation ou le refus des demandes de dialogue. Suivant les valeurs de ces paramètres, les demandes de dialogue seront acceptées ou rejetées. Ces paramètres peuvent

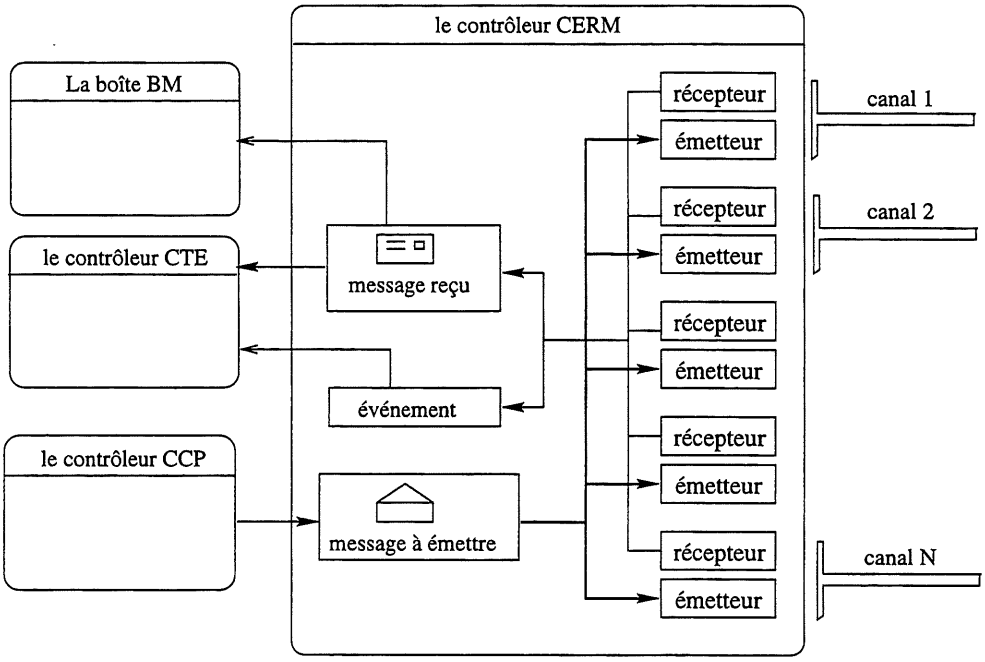

Figure 4.2 - *Contrôle d'émission et de réception de messages*

être mis à jour par le contrôleur CCP.

4.2.2 Le contrôleur CERM

La liste des tâches effectuées par le contrôleur CERM [figure 4.2] se résume en :

- L'établissement d'un dialogue (canal de communication) avec un agent serveur.
- L'écoute continue de tous les canaux de communication déjà établis entre un agent et ses homologues. Pour chaque canal, deux processus indépendants et concurrents lui sont affectés. L'un pour l'écoute, donc pour la lecture des messages parvenus sur le canal et l'autre, pour l'émission des messages sur celui-ci.

- La signalisation au contrôleur CTE de la réception d'un événement externe. Une liste des événements sera présentée dans le paragraphe réservé pour le contrôleur CTE. La signalisation de réception de nouveaux messages peut se faire de deux manières différentes suivant les paramètres du contrôleur CERM :

  1. Les nouveaux messages reçus seront traités dès leur réception l'un après l'autre suivant la loi du premier arrivé premier servi.
  2. Les nouveaux messages reçus seront stockés dès leur réception dans la boîte des messages.

- La signalisation à la boîte BM de la réception d'un nouveau messages. Dans le cas où l'option de traitement de messages dès leur réception n'est pas choisie, les messages reçus par le contrôleur CERM seront stockés dans la boîte BM.

- L'émission ou la diffusion de messages à une ou plusieurs destinations.

- La rupture d'un dialogue qui consiste à arrêter l'exécution des processus émetteur et récepteur du canal de communication relatifs au dialogue. Ces derniers ferment le canal de communication avant l'arrêt de leur traitement. Pour la rupture d'un dialogue, il y a deux manières de le faire :

  1. Rompre le dialogue sans attendre la confirmation de l'agent homologue. Dans ce cas, le canal de communication relatif au dialogue sera fermé.
  2. Envoyer un message à l'agent homologue pour demander la rupture du dialogue. Le canal de communication sera fermé dans le cas d'une réponse positive. Dans le cas contraire, le canal persistera ouvert. En cas d'absence de réponse à la demande de rupture de dialogue, le canal sera automatiquement fermé après un certain temps d'attente.

- La confirmation de l'acceptation ou du refus de la demande de rupture de dialogue avec l'agent homologue.
- La suspension temporaire d'un dialogue. Pour cela, un message sera envoyé à l'interlocuteur pour signaler la suspension temporaire du dialogue. Après la suspension d'un dialogue, tous les messages reçus par l'agent seront ignorés.
- La reprise d'un dialogue suspendu par l'envoi d'un message à l'interlocuteur pour signaler la reprise du dialogue suspendu.
- Confirmer la réception d'un message par le renvoi d'un accusé de réception à l'agent homologue.
- Aviser un agent distant d'un changement d'adresse.

Comme nous pouvons remarquer, le contrôleur CERM est supposé être capable d'exécuter plusieurs tâches simultanément. En effet, la lecture d'un message sur un canal de communication ne doit pas bloquer l'émission de messages ou l'écoute des autres canaux. D'où l'architecture qu'on propose de ce processus dans la figure 4.2. Dans le cas où un agent communique avec plusieurs autres agents simultanément, le processus CERM sera composé de plusieurs paires de processus (émetteur et récepteur) indépendants et concurrents affectés à chaque canal de communication.

Processus récepteur d'un canal PR

Le rôle de ce processus est de se mettre continuellement à l'écoute d'un canal de communication. Les messages lus sur ce canal seront signalés au contrôleur CERM. En cas de

problème de lecture sur le canal, un événement de rupture de canal sera signalé au contrôleur CERM qui à son tour signale le même événement au contrôleur CTE.

Processus émetteur sur un canal PE

Le rôle de ce processus est d'émettre des messages sur un canal de communication. Ce processus s'exécute en concurrence avec le processus récepteur du même canal.

4.2.3 La boîte de messages BM

Le rôle principal de la boîte des messages est de:

- Temporiser les messages récupérés par le contrôleur CERM. Les nouveaux messages sont insérés dans une file d'attente avec priorité. Par la suite, à la demande de l'utilisateur, ces messages seront récupérés de la file pour leurs traitements. Après avoir récupéré un nouveau message de la file, ce dernier sera supprimé de celle-ci. Suivant les paramètres de la boîte de messages, une copie des messages supprimés de la file sera sauvegardée dans des listes. Chaque liste correspond aux messages reçus sur un même canal. Nous présentons dans la figure 4.3 la structure générale de BM.
- Attendre la provenance d'un message sur un canal.
- Signaler la réception d'un message attendu au contrôleur CCP.
- Fournir la liste des messages reçus sur un canal.
- Déterminer le canal sur lequel un message est reçu.

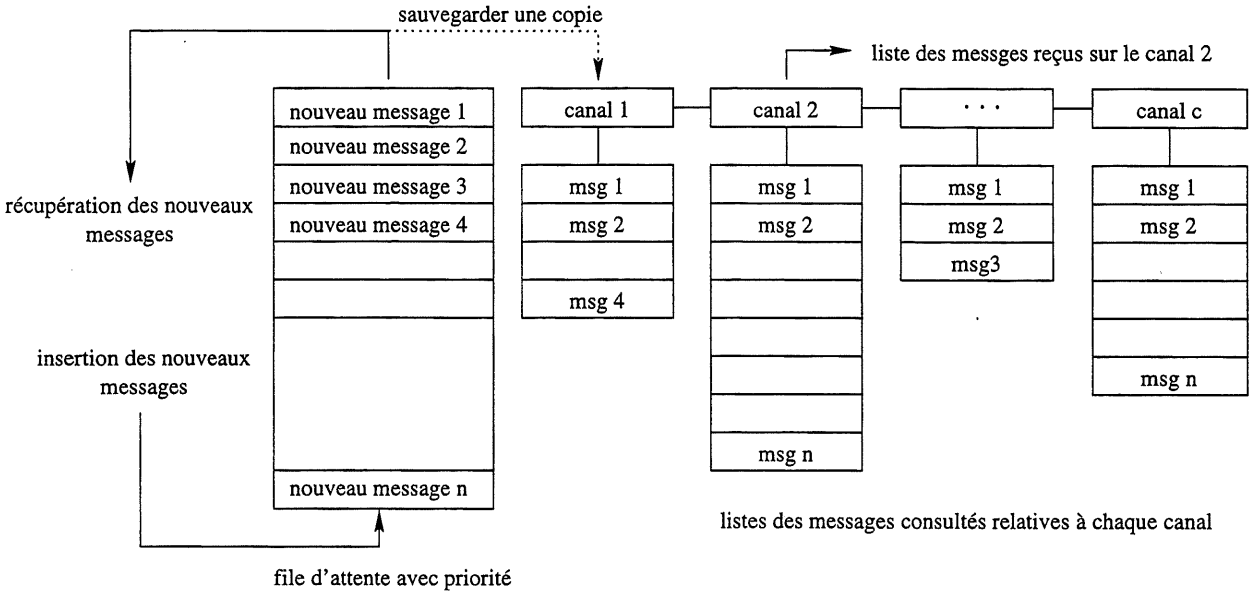

Figure 4.3 - *Boîte de messages*

- Consulter ou supprimer un message de la boîte de message.

4.2.4 Le contrôleur CTE

Cette entité consiste à traiter tous les événements externes signalés par le contrôleur CERM. La liste de ces événements est la suivante:

- La réception d'un nouveau message
- La réception d'un accusé de réception relatif à un message envoyé
- La réception d'une demande de rupture de dialogue
- La détection de rupture de dialogue

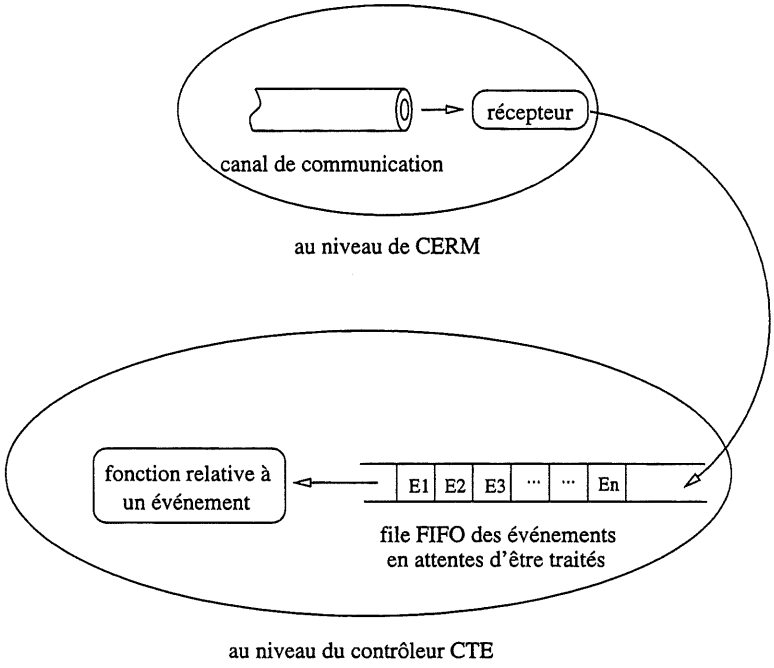

Figure 4.4 - *Traitement des événements externes*

- La réception d'un avis de suspension de dialogue
- La réception d'un avis de reprise d'un dialogue suspendu
- La réception d'un avis de changement d'adresse

Une fonction est associée à chaque événement. À chaque fois qu'un événement se produit, la fonction correspondante sera exécutée avec l'événement en question en paramètre. L'utilisateur doit alors redéfinir ces fonctions pour exprimer la manière dont ces événements doivent être traités. L'exécution de ces fonctions sera synchronisée pour éviter des confusions dans le cas où plusieurs événements de même type se produisent simultanément. Les événements seront traités alors l'un après l'autre suivant leurs ordres d'arrivée [figure 4.4].

4.2.5 Le contrôleur CCP

Le contrôleur CCP est conçu comme une interface entre la classe *agentCommunicant* et les autres entités présentées ci-dessus. Ainsi, cette interface permet à la classe `agentCommunicant` d'accéder aux différentes fonctionnalités de ces entités. Le contrôleur CCP regroupe toutes les fonctions de communication effectuées par un agent. La liste des fonctions définies dans cette entité est la suivante :

- Réserver un port de communication
- Libérer un port de communication
- Demander l'établissement d'un dialogue avec un agent serveur.
- Rompre un dialogue
- Répondre à la demande d'une rupture de dialogue
- Envoyer un message avec ou sans accusé de réception vers une ou plusieurs destinations.
- Confirmer la réception d'un message
- Attendre la réception d'un ou de plusieurs messages
- Récupérer les nouveaux messages sauvegardés dans la boîte BM
- Consulter ou supprimer un message de la boîte BM
- Déterminer le canal relatif à un message reçu
- Suspendre un dialogue

- Reprendre un dialogue suspendu
- Signaler un changement d'adresse
- Changer les paramètres de la boîte BM
- Changer les paramètres du contrôleur CERM
- Changer les paramètres du contrôleur TDD

4.2.6 La zone de représentation ZRCC

Cette partie est réservée à l'utilisateur pour inclure des nouvelles fonctions pour représenter les croyances de l'agent et son comportement. Cette partie de la classe `agentCommunicant` a un accès direct sur les fonctions des contrôleurs CCP et CTE. En effet, la deuxième partie du projet de développement d'outil générique pour les SMA consiste à enrichir cette partie par des mécanismes de représentation de connaissance et de comportement.

4.3 Structure d'un message

Dans une communication entre agents, il est important de structurer les messages échangés entre ces derniers. En effet, en recevant un message, un agent doit être capable de détecter l'adresse de provenance du message et l'identité de l'agent expéditeur. De plus, l'agent doit être capable de localiser et déchiffrer l'information utile dans le message émis par son agent homologue. D'autres informations peuvent être incluses dans un message tel que la priorité d'un message qui indique le niveau d'importance de celui-ci, le type de message qui sert à classer les messages en catégories et le type d'émission (avec ou sans accusé de réception).

Cependant, le problème qui se pose est au niveau de la représentation de l'information utile dans un message.

Actuellement, dans la plupart des outils de communication, le type de donnée réservé à l'information utile est prédéfini. Généralement ce type de donnée est un tableau d'octets ou une chaîne de caractères. Avec ces types de représentation, l'information utile doit subir une transformation avant de la copier dans le message à émettre (codage) ou avant de traiter le message reçu (décodage). Dans la figure 4.5 nous présentons un exemple de transformation de l'information avant son émission et après sa réception. Pour rendre l'outil OCA plus flexible et plus simple, nous offrons à l'utilisateur la possibilité d'émettre des informations de type quelconques. Cependant, ce choix a été pris après avoir fixé Java comme langage de programmation pour la mise en œuvre de l'outil OCA. Java offre des moyens simples pour émettre ou recevoir des objets de type quelconque. La structure générale d'un message est présentée dans la figure 4.6.

4.4 La synchronisation des ressources

Avec une architecture multi-processus telle que nous avons proposé pour l'outil OCA, et pour assurer le bon fonctionnement de celui-ci, une étude s'impose sur la synchronisation des processus et la gestion de partage de ressources. L'outil comprend sept types de processus : l'émetteur de message, le récepteur de message, la boîte de messages, le détecteur des connexions externes, le contrôleur d'émission et de réception de messages et la classe `agentCommunicant`. Nous pouvons distinguer deux types de ressource partageable :

- Une ressource de type donné qui pourrait être un objet de stockage de données.

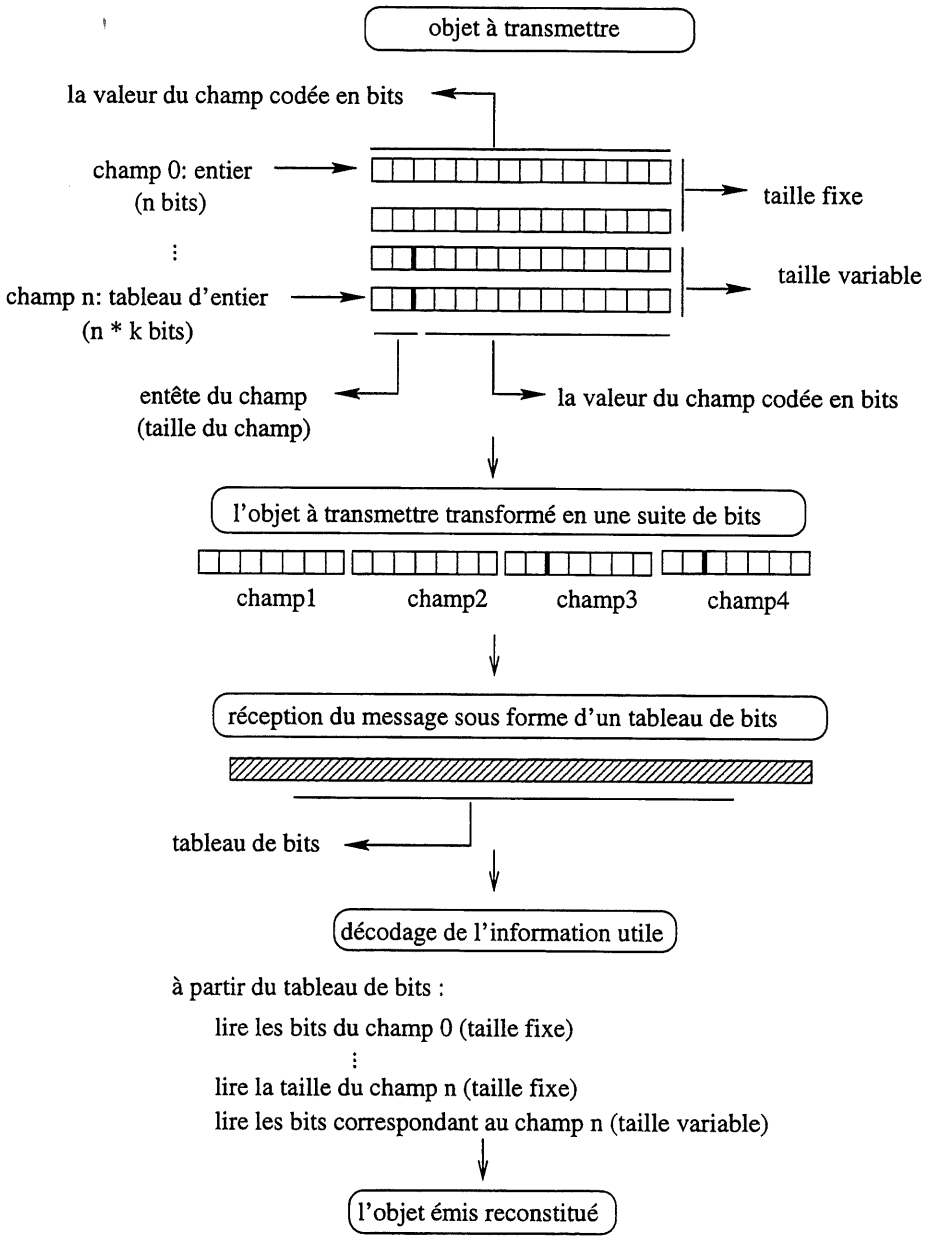

Figure 4.5 - *Codage et décodage d'un message à l'émission et à la réception*

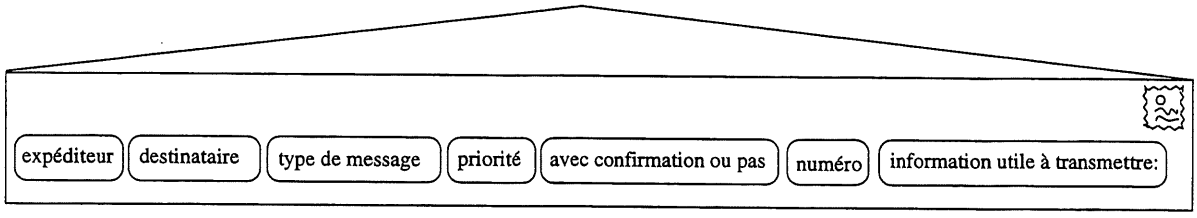

Figure 4.6 - *Structure d'un message*

- Une ressource de type tâche qui pourrait être une fonction ou une partie de fonction regroupant un ensemble d'instructions.

Les ressources communes de type données sont accédées en lecture ou en écriture. Une ressource de type tâche ne peut être accédée qu'en mode exécution. Cependant, plusieurs lectures simultanées sur un même objet ne causent aucun problème puisque toutes les lectures retourneront une même valeur et l'objet restera intact durant ce temps là. Par contre, plusieurs lectures et écritures concurrentes sur un même objet causeront des problèmes de cohérence et de conflit de données. Par exemple, deux processus émetteurs ne peuvent écrire simultanément sur le même canal de communication pour éviter les problèmes d'ambiguïté à l'autre bout du canal lors de la récupération des données émises.

Ainsi, suivant les cas, plusieurs accès simultanés à un même bloc de traitement peut causer des conflits. Par exemple, un conflit de traitement peut se produire si jamais deux processus d'écoute signalent simultanément l'arrivée de deux nouveaux messages en faisant appel à une même fonction du contrôleur CTE. D'où la nécessité de gérer et de synchroniser l'accès aux données et l'accès aux fonctions. La discipline d'accès aux ressources de type tâche que nous proposons est celle du premier arrivé premier servi. Les ressources à synchroniser sont les suivants :

- Tous les canaux de communication.

- La boîte de message.
- Toutes les fonctions de traitement des événements dans le contrôleur CTE
- La fonction d'attente de message.

4.5 Mise en œuvre

4.5.1 Choix du langage de programmation

Après avoir défini l'outil OCA sur les plans structurels et fonctionnels, nous devons passer à l'étape de développement. La première question que nous nous sommes posée après avoir fini l'étape de conception était la suivante: Quel sera le meilleur outil ou langage de programmation pour la mise en œuvre de cet outil?

L'étude bibliographique sur les outils multiagents présentée dans le chapitre 2 nous a beaucoup aidés sur le choix du langage de programmation. En effet, les premières plates-formes pour les SMA étaient implémentées à l'aide de langages standard (souvent lisp), mais les plates-formes récentes utilisent la programmation par objets.

La programmation par objet est l'outil le plus naturel pour implémenter les agents. L'objet, élément de base de la programmation par objets, peut être considéré utilement comme une brique de base pour l'implémentation de modèles d'agents. La liste des langages par objet est longue, parmi celle-ci nous citons les candidats les plus intéressants, vis-à-vis nos besoins, tel que Smalltalk, C++ et Java. Notre choix a été fixé sur le langage Java. Nous proposons dans ce qui suit une présentation générale du langage choisi en comparaison avec C++ et Smalltalk

Smalltalk, C++ et Java

Quelque part entre Smalltalk et C++, Java est le dernier-né des langages de programmation par objets. Reprenant le principe Smalltalk d'indépendance par rapport au support d'exécution, Java repose sur un pseudo-code interprété par une machine virtuelle [4]. D'une certaine façon, on pourrait présenter Java comme le langage intégrant le meilleur des deux mondes C++ et Smalltalk. C'est le point de vue des auteurs de ce langage [26] et d'un nombre important de programmeurs qui ont rapidement adopté ce nouveau langage.

D'un autre côté, on pourrait voir Java comme un compromis entre deux approches diamétralement opposées de la programmation par objets. L'une est fortement statique basée sur un système de typage (C++) et l'autre est fortement dynamique sans contrôle de type (Smalltalk). La notion de compromis sous-tend l'idée d'une limitation des capacités vis-à-vis des approches plus radicales.

Nous présentons dans ce qui suit un tour d'horizon rapide de Java qui peut être effectué par quelques points de comparaison avec Smalltalk et C++. Sur beaucoup d'aspects, il se situe entre ces deux langages :

- La syntaxe : Très proche de C++ mais plus réduite, elle demeure cependant plus verbeuse que la syntaxe minimaliste de Smalltalk. Gosling [26] décrit Java comme : C++ sans les couteaux ni les revolvers. En effet, Java est un dérivé simplifié de C++. Un travail d'amaigrissement a été réalisé et concerne les éléments les plus complexes et sous utilisés de C++ ;
- À base de classes : De même que Smalltalk, tout est objet et l'invocation des méthodes est dynamique (fonctions membres virtuelles de C++). La classe constitue l'unique

espace de définition. Contrairement à C++, il n'existe pas de variables globales ni de fonctions définies hors des classes. Les classes Java comme les classes Smalltalk dérivent toutes d'une racine commune, la classe `Object` ;

- Une ramasse-miettes (garbage collector) : De même que Smalltalk, Java libère le programmeur de la restitution explicite de la mémoire occupée par des objets qui ne sont plus référencés. Notons qu'à la différence de C++, il n'y pas de pointeur et aucune manipulation explicite d'adresses n'est possible ce qui veut dire pas d'allocation de mémoire ni de libération de mémoire à gérer.

- Une machine virtuelle d'exécution : De même que Smalltalk, l'exécution d'un programme Java est basée sur un code généré indépendamment de toutes architectures. Le compilateur Java ne génère pas des instructions machines spécifiques mais un programme en byte-code, qui peut être décrit comme un langage machine pour un processeur virtuel qui n'a pas d'existence physique. Ce code objet compilé peut être alors exécuté par un interpréteur Java qui n'est ni plus ni moins qu'un émulateur de processeur virtuel : la Machine Virtuelle Java.

- L'héritage : De même que Smalltalk, Java impose l'héritage simple pour l'héritage classique de structure et de comportement. Cependant, de façon similaire au langage IDL de la spécification CORBA [27], Java introduit une notion d'interface, ce qui permet de distinguer l'héritage de structure de l'héritage d'interface. Pour ce dernier, Java permet l'héritage multiple qui ne pose aucun problème de conflit de noms.

- Une bibliothèque de classes significatives : De même que Smalltalk, le développement d'applications en Java repose largement sur l'existence d'une hiérarchie de classes de base. Ainsi, l'environnement JDK (Java Developers Kit) [44] fournit une base de développement en Java.

- Les paquetages (package): Les paquetages définissent un espace de nommage structuré de façon arborescente. Cette organisation reprend l'organisation des systèmes de fichiers des systèmes d'exploitation. C'est un plus par rapport aux versions actuelles de Smalltalk qui ne permettent pas de définir plusieurs classes de même nom.

En plus de ces concepts de base, Java offre un certain nombre de mécanismes tels que :

- Les processus légers (threads) qui permettent de gérer plusieurs flots de contrôle au sein d'une même application. Dans les programmes multithréadés, plusieurs processus peuvent s'exécuter simultanément. Java inclut le support des processus légers multiples allégeant considérablement l'écriture de programmes qui utilisent ces fonctionnalités. Le langage contient un jeu de primitives de synchronisation basé sur les travaux de Hoare [2].

- Synchronisation : Pour assurer le comportement correct d'un programme qui s'exécute dans un environnement multi-thread, il est nécessaire d'employer les mécanismes qui imposent un accès "sûr" au code du programme. Java gère ceci en deux volés :

  1. Tous les accès mémoire pour toutes les variables, autre que long et double, sont atomiques. Ceci signifie que quand il y a plusieurs mises à jour concurrentes sur une même variable, une lecture concurrente sur cette variable retournera une des valeurs écrites, ou une ancienne valeur, mais jamais une valeur intermédiaire ou probablement incorrecte.
  2. Java supporte l'utilisation des moniteurs pour permettre l'accès séquentiel aux sections de code bien déterminées. Java met en application un mécanisme de moniteur dans chaque objet ou classe de Java. Essentiellement chaque objet est associé à

une serrure simple qui est implicitement verrouillée et déverrouillée avant et après chaque accès.

- Sécurité : Qui dit réseau, implique risque, paranoïa, virus et infiltrations, de plus il est courant pour une application distribuée de télécharger une classe Java distante pour son exécution locale. Donc, assurer la sécurité des programmes qui circulent sur le réseau, est un problème majeur. Java a intégré, dès la conception, plusieurs mécanismes de sécurité visant à rendre les programmes fiables et à éliminer les risques de virus par la vérification de la légitimité de leur code.

- Dynamique : Les bibliothèques externes de Java qui offrent ces possibilités ainsi que les bibliothèques écrites par le développeur ne sont pas gérées de la même manière qu'en C++. Il n'est pas nécessaire de modifier le programme si une de ses bibliothèques change, les classes sont chargées dynamiquement. Ainsi Java résout le problème classique de C++ nommé le problème de superclasse fragile. Certaines portions de code d'une application peuvent être modifiées au cours de l'exécution d'une application Java. En particulier, une application peut utiliser du code provenant d'un site serveur distant accédé via le Web. Java offre un mécanisme de chargement et d'édition des liens dynamiques permettant de charger du code en cours d'exécution.

- Distribué : Les appels aux fonctions d'accès réseau (sockets) et les protocoles Internet les plus utilisés tels HTTP, FTP, Telnet sont intégrés dans Java. On peut écrire très simplement une architecture client-serveur sous Java et utiliser des fichiers sur des systèmes distants comme s'ils se trouvaient sur un disque dur local.

- Portable : Java respecte de nombreux standards tant au niveau réseau, qu'au niveau interne ou d'interface. En effet dans Java, les types sont identiques quelle que soit l'implémentation et le matériel (Alpha 64 bits à 250 MHz ou 68030, 25 MHz 16/32 bits).

Java définit explicitement dans "The Language Specification" de Sun, qu'un "Int" est toujours un entier de 32 bits en complément à deux signés. Les formats numériques respectent la norme IEEE 754 et les chars la norme Unicode.

- Performance : Malgré qu'on nous apprend que les interpréteurs sont lents, lourds et gourmands, l'interpréteur de Java échappe à cette règle puisqu'il est plus qu'un interpréteur. Il fait tourner un programme généré par un compilateur optimisé sur une machine virtuelle. On n'atteint pas les performances du C ou C++ mais l'écart est de plus en plus réduit avec les nouvelles techniques d'optimisation et les JIT.

Pour compléter cette étude comparative, nous allons présenter également des aspects de C++ et Smalltalk qui ne sont pas présents dans Java. Parmi ceux-ci, nous pouvons déjà citer :

- La généricité : Java ne dispose pas du mécanisme de généricité (les templates) qui caractérise le typage et la génération de code fortement statiques de C++ ;
- les fermetures lexicales : À l'opposé de SmallTalk, Java ne dispose pas non plus des fermetures lexicales (BlockClosure) qui permettent d'exprimer les différentes structures de contrôle du langage.

Justification du choix

Dans le choix du langage de programmation nous avons fixé des critères de sélection. Ces critères sont en rapport direct avec les spécifications de l'outil OCA. Nous présentons dans

ce qui suit la liste de ces critères avec les fonctionnalités correspondantes.

- Orienté objet : Un langage orienté objet est l'outil le plus naturel pour implémenter les agents. Un objet peut correspondre à un modèle d'agent.
- Portable : Parmi les spécifications de l'outil OCA nous avons précisé l'indépendance de celui-ci des machines et des systèmes d'exploitation utilisés.
- Multitâche : Un agent doit être en mesure d'exécuter plusieurs traitements en même temps.
- Distribution : Les agents formant un SMA ou un système distribué peuvent être logés sur des machines distinctes et distantes, liées par un réseau de communication.
- Synchronisation : Nous avons déjà discuté dans le paragraphe 4.4 le problème d'accès aux ressources et la nécessité d'une synchronisation pour résoudre ce problème.
- Sécurité : Qui dit communication sur un réseau public dit risque des virus et des infiltrations.
- Performance : Les performances de l'outil dépendront en partie de celles du langage avec lequel il a été développé.

Malgré sa lenteur d'exécution, Java est le langage qui répondait le plus à nos besoins. En effet, en comparaison avec le langage C++, Java est dix fois moins rapide que ce dernier. Ce point faible de Java est expliqué par le fait que les programmes compilés (le cas de C++) ont toujours une tendance à être plus rapide en exécution que les programmes interprétés (le cas de Java ou Smalltalk). Comme Java n'est qu'un nouveau langage qui vient juste de naître, et que plusieurs recherches se poursuivent sur l'optimisation de la machine virtuelle

de celui-ci, nous devrons nous attendre à une amélioration de ces performances sur les plans temps d'exécution et ressources utilisées.

4.5.2 Réalisation

Dans la figure 4.7 nous présentons les différentes classes développées en Java formant l'outil OCA. Ces classes peuvent être seindées en deux catégories:

- Les classes publiques (accessibles directement par l'utilisateur de OCA) :
  - La classe principale `agentCommunicant`
  - La classe `mail`
- Les classes privées (non accessibles par l'utilisateur de OCA) :
  - La classe `detectConnection`
  - La classe `controlSendReceiveMessage`
  - `sendMessage`
  - `receiveMessage`
  - `mailBox`

Les classes privées étaient conçues pour répondre aux besoins de la classe principale *agentCommunicant* et pour permettre le traitement concurrent et multitâches dans l'outil OCA. C'est pourquoi toutes les classes présentées dans la figure 4.7 sont dérivées de la classe `Thread` de Java. La classe `agentCommunicant` regroupe les trois dernières parties de la structure générale (le contrôleur CTE, le contrôleur CCP et la zone ZRCC) de l'outil OCA présenté dans le paragraphe 4.2. Les classes `detectConnection`, `controlSendReceiveMessage`,

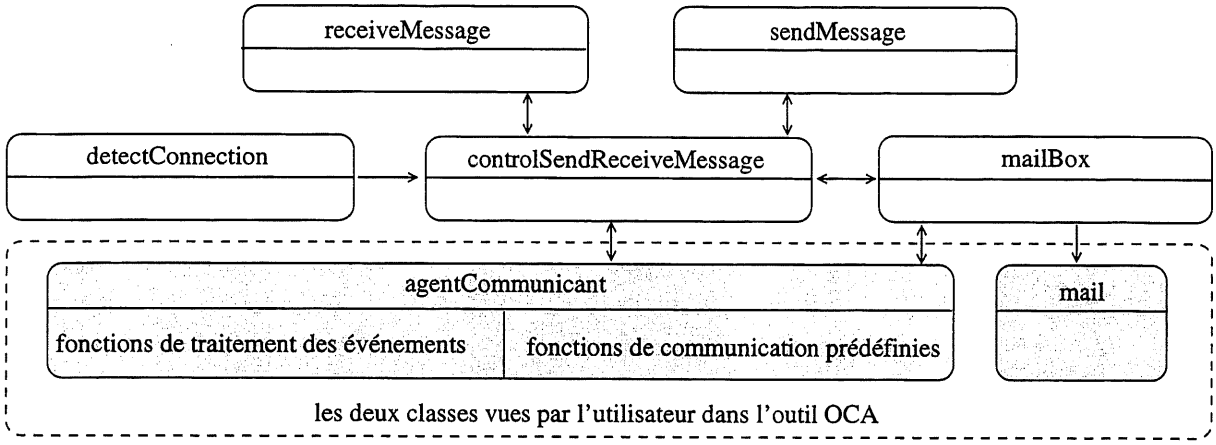

Figure 4.7 - *La librairie de classes formant l'outil OCA*

`sendMessage`, `receiveMessage` et `mailBox` correspondent respectivement au processus TDD, contrôleur CERM, processus émetteur de messages, processus récepteur de messages et la boîte BM.

La classe detectConnection

Cette classe représente le processus de traitement des demandes de dialogue pour un agent serveur (TDD). Dans le cas d'un agent serveur, la classe `agentCommunicant` fait appel à cette classe pour créer un port de communication et pour détecter et traiter toutes les demandes de dialogues des autres agents. Après chaque connexion externe sur le port de communication, la classe `detectConnection` signale à la classe `controlSendReceiveMessage` (contrôleur CERM) la détection d'une nouvelle demande de dialogue. Cette dernière lance deux processus de type `sendMessage` (processus émetteur) et `receiveMessage` (processus récepteur) pour émettre et recevoir des messages sur le canal établi. La libération du port de communication se fait au niveau de cette classe. À la suite de fermeture de port de communication, l'agent ne pourrait plus être sollicité pour un dialogue, donc il ne sera plus considéré comme un agent

serveur.

Les classes receiveMessage et sendMessage

Comme nous l'avons précisé dans le paragraphe précédent, le rôle de la classe `receiveMessage` est de surveiller continuellement l'arrivée des messages sur un canal de communication. La classe `receiveMessage` avise la classe `agentCommunicant` de la réception des messages sur un canal, en faisant appel à la fonction `haveMail` de celle-ci. La fonction `suspendDialog` permet de suspendre momentanément le processus de lecture des nouveaux messages sur un canal de communication et la fonction `closeDialogue` permet de l'arrêter complètement, ce qui veux dire la fermeture du canal de communication. La reprise de la lecture des messages se fait par l'appel de la fonction `resumeSuspendedDialog`. Le rôle de la classe `sendMessage` est d'émettre des messages sur le canal de communication relative à cette classe.

La classe controlSendReceiveMessage

Cette classe consiste à gérer l'ensemble des couples de processus récepteur et émetteur de messages relatives aux canaux de communication. Cette classe correspond en fait au contrôleur CERM. Les fonctions offertes par cette classe sont les mêmes fonction présentées dans le contrôleur CERM.

La classe mailBox

Cette classe représente la boîte de messages BM. En effet, suivant les paramètres de cette classe, un nouveau message passe directement à la fonction `haveMail` de la classe

`agentCommunicant` pour son traitement où il sera temporisé dans la classe `mailBox` en attendant son traitement. Les messages sont temporisés dans une file d'attente de type premier arrivé premier servi avec priorité [figure 4.3]. Par la suite, ces messages seront retirés de la file en faisant appel à la fonction `getNewMessage` de la classe `agentCommunicant`. Suivant la configuration de la classe `mailBox` les messages retirés de la file peuvent être sauvegardés dans la boîte de messages. Ces messages pourront être consultés ou supprimés à partir de la classe principale. La fonction `getCanalMessage` de la classe `mailBox` permet de déterminer la provenance (le canal de réception) d'un message. La fonction `getMessagesCanal` permet de déterminer l'ensemble des messages reçus sur un canal quelconque. Les messages et les canaux de communication sont identifiés par des entiers positifs.

La classe mail

Nous pouvons considérer la classe mail comme étant une enveloppe contenant l'information utile à émettre ou celle qui était reçue. Avant d'émettre une information à un agent, l'utilisateur doit créer une enveloppe, mettre cette information dans l'enveloppe et finalement appeler la fonction `sendMessage` de la classe `agentCommunicant` (l'enveloppe en paramètre) pour émettre le message. La structure de la classe mail est présentée dans la figure 4.6. Les attributs de la classe mail sont :

- `public Object message` : l'information utile à émettre.
- `public int canal` : le numéro de canal sur lequel le message sera émis ou sur lequel le message est reçu.
- `public int type` : le type de message dont la valeur par défaut est 0.

- `public int priority`: le niveau de priorité de message qui prend la valeur 0 par défaut.
- `public String from`: l'adresse électronique de l'expéditeur. La valeur par défaut est `null`.
- `public String to`: l'adresse électronique du destinataire. La valeur par défaut est `null`.
- `public boolean accuse`: détermine si un accusé de réception est requis ou pas. La valeur de cet attribut est `false` par défaut.
- `public int number`: ce champ correspond au numéro d'un message émis ou reçu.

Cependant, au moins les deux premiers attributs doivent être remplis pour assurer l'arrivée d'un message à sa destination. Les autres attributs sont facultatifs. L'instantiation d'un objet de type `mail` se fait via la commande `new`. Deux types de constructeur sont définis. Le premier prend en paramètre les deux premiers attributs, le deuxième nécessite tous les attributs. Exemples de création d'objet de type `mail`:

```
int canal=5 ; // le numéro de canal sur lequel le message sera émis
String data="Ceci est un objet de type String à émettre";
mail message1=new mail(canal,data);
hashTable hash= new hashTable(); // l'information utile à émettre.
int type=3;
int priorite=4;
String from="Etudiant:pollux.gel.usherb.ca";
String to="Professuer:callisto.si.usherb.ca";
```

```
Boolean accuse=true;
mail message2=new mail(canal,hash,from,to,type,priorite,accuse);
```

La valeur du champ number est déterminée par la classe controlSendReceiveMessage.

La classe principale agentCommunicant

La classe agentCommunicant est la classe la plus importante vis-à-vis l'utilisateur de l'outil OCA. En effet, la mise en œuvre d'un agent communicant avec l'outil OCA consiste à la création d'une classe dérivée (sous-classée) de la classe agentCommunicant qui représente un modèle de base d'agent communicant. La classe dérivée hérite de la classe agentCommunicant toutes les fonctions de communication prédéfinies dans celle-ci. Nous rappelons que la classe agentCommunicant est constituée par le contrôleur CCP, le contrôleur CTE et la zone de représentation ZRCC. La tâche de l'utilisateur consiste alors à habiller cette classe de base par l'ajout de fonctions et de données, relatives à l'application multiagents, dans la zone ZRCC. La mise en œuvre de ces fonctions peut faire appel aux fonctions prédéfinies dans la classe agentCommunicant correspondant aux fonctions relatives au contrôleur CCP. Pour le traitement des événements externes, l'utilisateur doit réécrire les fonctions prédéfinies dans le contrôleur CTE. Nous présentons dans ce qui suit la liste des fonctions définies dans la classe agentCommunicant. Nous commençons par la présentation des fonctions formant le contrôleur CTE présenté ci-dessus suivi par les fonctions relatives au contrôleur CCP.

Les fonctions relatives au contrôleur CTE : Toutes les fonctions présentées ci-dessous peuvent être réécrites par l'utilisateur pour le traitement des événements correspondants.

- `void haveMail(mail Msg)` : Dans le cas où l'option de traitement de messages dès leur réception est choisie, cette fonction sera appelée automatiquement à la réception de chaque nouveau message sinon le message sera stocké dans la boîte de messages.
- `void okMessageReceived(int numMessage)` : Cette fonction permet de signaler la réception d'un accusé de réception d'un message identifié par `numMessage`.
- `void dialogIsSupended(int numCanal)` : Cette fonction permet de signaler la suspension d'un dialogue
- `void suspendedDialogIsResumed(int numCanal)` : Cette fonction permet de signaler la reprise du dialogue sur le canal identifié par `numCanal`.
- `void canalIsClosed(int numCanal)` : Cette fonction est appelée automatiquement à la suite d'une détection de fermeture de canal.
- `void closeDialogRequest(int numCanal)` : Cette fonction est exécutée à chaque réception d'une demande de fermeture de dialogue.
- `void dialogueIsClosed (int numCanal)` : Cette fonction est appelée pour signaler la rupture d'un dialogue.
- `void changeOfAddress(int numCanal)` : Cette fonction permet de signaler la réception d'un avis de changement d'adresse d'un agent
- `void okChangeOfAdress(int numCanal)` : Cette fonction signale la réception d'une confirmation de réception d'un avis de changement d'adresse.

Les fonctions relatives au contrôleur CCP : Les fonctions présentées dans cette partie sont prédéfinies dans la classe `agentCommunicant`. L'utilisateur peut se servir directement de ces fonctions pour implémenter les tâches de communication d'agent.

- `boolean openPort(int numPort)` : Cette fonction permet d'activer le processus de traitement des demandes de dialogue pour servir d'autres agents TDD. Un port de communication de numéro `numPort` sera alors réservé sur la machine sur laquelle l'agent s'exécute. La fonction retourne la valeur `true` si le port a été créé avec succès et elle retourne la valeur `false` dans le cas contraire. La réservation d'un port peut échouer dans les cas suivants :
  - Il existe déjà un port ouvert pour l'agent en question.
  - Le port en question est déjà réservé par une autre application.
  - Le système ne permet pas à l'utilisateur de créer un port de communication.
- `int conectAgent(String adresseIP, int port)` : Cette fonction permet de demander le dialogue à un agent serveur identifié par son adresse IP et par le numéro de son port de communication. La valeur de retour de cette fonction peut avoir plusieurs significations :
  1. La valeur 0 : le rejet de la demande par l'agent homologue.
  2. La valeur -1 : l'adresse IP ou le numéro de port sont erronés.
  3. La valeur -2 : la présence d'un problème de réseau.
  4. Une valeur ≥ 1 : l'acceptation de la demande et la valeur retournée représente le numéro du canal établis.

- `boolean closePort()` : cette fonction ferme et libère le port de communication réservé par la fonction `openPort()`. La valeur de retour de cette fonction est `true` si l'opération est effectuée avec succès. Dans le cas contraire (absence de port ouvert) la fonction retournera la valeur `false`.

- `boolean closeDialogue(int numCanal, boolean confirmation)` : L'appel de cette fonction permet de mettre fin à un dialogue avec ou sans une confirmation de l'agent homologue. La valeur du paramètre confirmation détermine si la rupture de dialogue est avec ou sans confirmation. Dans le cas d'une rupture de dialogue sans confirmation, le canal de communication sera fermé et la fonction retournera la valeur `true`. Dans le cas contraire la valeur de retour sera:

  - `true`: si la demande de rupture est confirmée par l'agent homologue. Dans ce cas le canal de communication sera fermé.
  - `false`: si la demande de rupture est rejetée par l'agent homologue. Dans ce cas le canal de communication persistera ouvert.

- `synchronized int sendMessage(mail message)` : Cette fonction permet d'émettre un message à un agent. La valeur du champ canal du message passé en paramètre déterminera le canal de communication sur lequel le message sera émis. Cette fonction est synchronisée pour éviter plusieurs transmissions simultanées sur le même canal, ce qui engendrera des confusions à la réception à l'autre bout du canal. La valeur de retour est un nombre positif si le message est envoyé avec succès. Cette valeur correspond au numéro du message émis et elle pourrait servir par la suite pour vérifier la réception d'un accusé de réception relatif au message émis. La fonction retourne la valeur 0 dans les cas suivants:

  - Le canal est fermé.

  - Le numéro de canal ne correspond à aucun canal.

- `synchronized int[] sendMessage(mail mg, String adresse, int port)` : Cette fonction effectue deux tâches en même temps. Elle consiste à :
  - Établir un dialogue, via la fonction `connectAgent` avec l'adresse et le port passé en paramètre.
  - Affecter le numéro du canal établi à l'attribut canal du message passé en paramètre.
  - Émettre le message passé en paramètre sur le canal établi via la fonction *sendMessage*.

  La valeur de retour de cette fonction est un tableau dont le premier élément correspond à la valeur de retour de la fonction `connectAgent` et le deuxième élément est valeur de retour de la fonction `sendMessage`.

- `synchronized int[] sendMessage(mail mg, int canaux[])` : Cette fonction permet de diffuser un message à un groupe de destinations. Un tableau d'entier passé à cette fonction pour identifier les numéros des canaux sur lesquels le message sera émis. La valeur de retour est un tableau d'entiers. Chaque élément du tableau correspond à la valeur de retour de la fonction `sendMessage` sur le canal correspondant.

- `mail waitMessage(int numCanal)` : L'appel de cette fonction suspendra l'exécution de l'agent jusqu'à la réception d'un nouveau message sur le canal identifié par `numCanal`. La fonction retournera le message attendu ou une valeur `null` dans le cas où le canal est fermé.

- `mail[] waitMessages(int[] numCanaux)` : Avec le même principe de la fonction *waitMessage*, la fonction `waitMessages` permet à un agent d'attendre plusieurs messages

provenant sur un ensemble de canaux. Les canaux de communication sont identifiés par un tableau d'entiers correspondants aux numéros de ces canaux. Le traitement de l'agent sera alors suspendu jusqu'à la réception de tous les messages attendus.

- `mail getNewMessage()` : Cette fonction permet de récupérer les nouveaux messages de la boîte de messages. Dans le cas où la boîte de messages ne contient aucun nouveau message, la fonction retournera la valeur `null`. Après avoir récupéré un nouveau message, et suivant les paramètres de la classe `mailBox`, ce dernier sera sauvegardé ou supprimé de celle-ci.

- `boolean suspendDialogue(int numCanal)` : Cette fonction permet de suspendre un dialogue. La valeur de retour est `false` dans le cas où le canal identifié par `numCanal` est fermé ou non existant.

- `boolean resumeSuspendedDialogue(int numCanal)` : Cette fonction permet de reprendre un dialogue suspendu. La valeur de retour est `false` dans le cas ou le canal identifié par `numCanal` est fermé ou non existant.

- `mail getMessage(int identMsg)` : Cette fonction permet de consulter un message dans la boîte de messages. Un objet de type mail sera retourné si le message est trouvé, ou une valeur `null` dans le cas contraire.

- `int getCanalMessage(int identMsg)` : Cette fonction permet de déterminer le numéro de canal sur lequel le message, identifié par son numéro, est reçu. La valeur de retour est 0 si le canal en question est fermé ou une valeur positive correspondant au numéro de canal.

- `boolean removeMessage(int Msg)` : Cette fonction supprime un message identifié par son numéro de la boîte de messages.

- `void setMailInMailBox(boolean val)` : Cette fonction permet de mette à jour les paramètres de la classe `controlSendReceiveMessage`. La valeur booléenne passée en paramètre permet de déterminer si les nouveaux messages reçus seront sauvegardés dans la boîte de messages ou traités par la fonction `haveMail`. Les nouveaux messages seront stockés, par défaut, dans la boîte de messages.

- `void setMailArchive(boolean val)` : Cette fonction permet de mette à jour les paramètres de la classe `mailBox`. La valeur booléenne passée en paramètre permet de déterminer si les messages récupérés de la boîte de messages seront sauvegardés ou supprimés de celle-ci. Par défaut, les messages récupérés de la boîte de messages seront supprimés de celle-ci.

- `void setDialogRequest(int etat)` : Cette fonction permet de mettre à jour les paramètres de la classe `detectConnection`. Le nombre entier passé en paramètre permet de déterminer si les demandes de dialogue seront acceptés (etat=1) ou rejetés (etat=2).

4.6 Exemples d'utilisation de l'outil OCA

Nous présentons dans ce paragraphe quelques exemples de programmes simples écrits en Java avec la librairie de l'outil OCA.

4.6.1 Simple échange de messages

Cette application consiste à faire un simple échange de messages de type chaîne de caractères entre deux agents distants. Le code source de l'application est le suivant:

```
//*****************************************
```

```
//* Le code source de la classe Serveur.class *
//*********************************************
import java.agentCommunicant
public class Serveur extends agentCommunicant{
public Serveur(){
  // réserver un port de communication
  openPort(5000);
  // activer le mode traitement direct des messages
  setMailInMailBox(false);
}
// redéfinir la fonction haveMessage pour traiter les messages reçus
public void haveMessage(mail m){
  Integer destination ;
  // dans cette application, les messages échangés sont de type String
  String message =(String)m.message ;// récupérer de l'information utile
  System.out.println("Serveur: j'ai reçu un message :"+message);
  if (message.equals("Bonjour")){
    destination=message.from; // récupérer la provenance du message
    mail reponse=new mail("Bonjour",destination);
    // réponse du serveur au message du client
    senMessage(reponse);
  }
  if (message.equals("qu'est ce que tu fais dans la vie ?");{
    destination=message.from;
    mail reponse=new mail("Je travaille à Sherbrooke",destination);
    senMessage(reponse);
  }
  if (message.equals("au reveoir")){
    int destination=message.from;
    mail reponse=new mail("bye bye ! et à la prochaine !",destination);
    senMessage(reponse);
  }
}
//*********************************************
```

```
//* Le code source de la classe Client.class *
//******************************************
import java.agentCommunicant
class Client extends agentCommunicant {
public void start(){
  mail bonjour=new mail("Bonjour");
  int canal=sendMessage(message,pollux.gel.usherb.ca,5000);
  if  (canal >0) {
    // attendre la réponse du serveur
    mail r1=waitMessage(canal);
    String reponse1=(String)r1.message;
    System.out.println("Client: la réponse du serveur est" + reponse1);
    mail q1=new mail("qu'est ce que tu fait dans la vie ?");
    if (sendMessage(q1,canal)){
      mail r2=waitMessage(canal);
      String reponse2=(String)r2.message
      System.out.println("Client : la réponse du serveur est" + reponse2);
      mail auRevoir=new mail("au revoir");
      String reponse3=(String)r3.message;
      if (sendMessage(auRevoir,canal)){
        mail r3=waitMessage(canal);
        System.out.println("Client : la réponse du serveur est" + reponse3);
      }
    }
  }
}
}
```

Pour démarrer l'application il faut lancer l'exécution de la classe `Serveur.class` avant celle de la classe `Client.class`. La classe `Serveur.class` doit être exécutée sur la machine d'adresse `pollux.gel.usherb.ca`. Dans le cas où cette classe est exécutée sur une autre machine, l'utilisateur doit remplacer l'adresse de destination (`pollux.gel.usherb.ca`) dans

le fichier `Client.java` par l'adresse IP de la nouvelle machine. L'exécution de la classe `Client.class` peut se faire sur la même machine que la classe `Serveur.class` ou sur une autre machine distante. L'affichage de cette application est comme suit :

```
Serveur: j'ai reçu un message: Bonjour
Client: la réponse du serveur est: Bonjour
Serveur: j'ai reçu un message: qu'est ce que tu fais dans la vie ?
Client: la réponse du serveur est: Je travaille à Sherbrooke
Serveur: j'ai reçu un message: au revoir
Client: la réponse du serveur est: bye bye ! et à la prochaine !
```

4.6.2 Calcul factoriel

Cette application consiste à calculer la valeur de N! par la collaboration de plusieurs agents de multiplication. Le principe de base du calcul consiste à décomposer les N-1 multiplications en plusieurs sous-ensembles de multiplications de tailles plus réduites que N-1. Chaque agent se charge de calculer le produit de chaque sous-ensemble. Un agent superviseur récupère les résultats de chacun des agents multiplicateurs et calcule le produit de ces résultats pour générer le résultat final de N! [figure4.8]. Il est clair que le temps de calcul de N! avec cette méthode est presque divisé par le nombre d'agents multiplicateurs. Cette conclusion est vraie si le nombre d'agents est bien choisi. Car si le nombre d'agents s'approche de N, le temps de calcul sera plus élevé que celui du temps de calcul avec un seul agent. Ceci est dû au temps pris par la communication entre les agents. Nous présentons dans ce qui suit un exemple de calcul fatoriel avec trois agents multiplicateur et un agent superviseur.

```
//*****************************************************
//* Le code source de la classe agentFactorielle.class *
//*****************************************************
import java.agentCommunicant
```

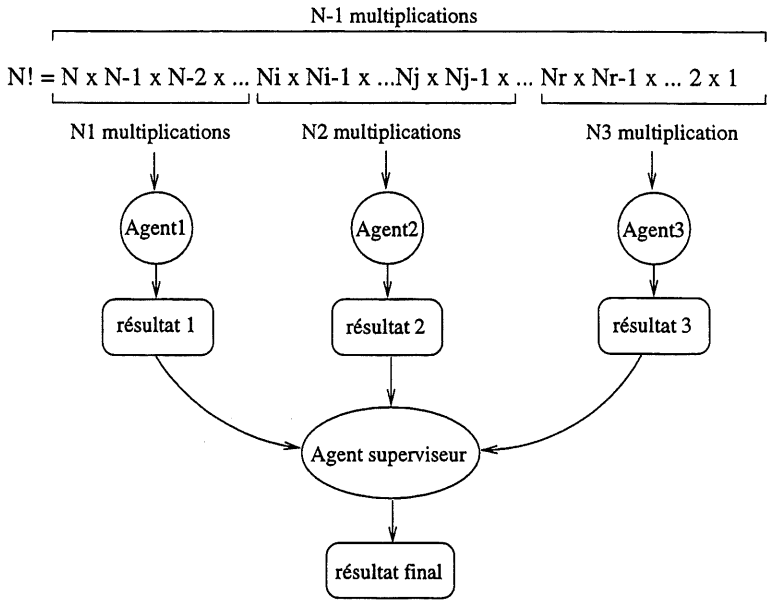

Figure 4.8 - *Principe de base du calcul distribué de N!*

```
public class agentFactorielle extend agentCommunicant{
int N;
int nbReponse=0;
long resultat=1;
String adresse1,adresse2,adresse3;
int port1,port2,port3;
public void agentFactorielle(int n,String a1,int p1,String a2,int p2,
                                                    String a3,int p3)
N=n; adresse1=a1; port1=p1; adresse2=a2; port2=p2; adresse3=a3; port3=p3;
}
public void start(){
  // représenter le premier intervalle par un vecteur
Vector interval1=new Vector();
  interval1.addElement(new Integer(1));
  interval1.addElement(new Integer(N/3))
  mail msg1=new mail(interval1,adresse1,port1);
  // demander à un agent multiplicateur de calculer le produit
```

```
sendMessage(msg1);
  Vector interval2=new Vector();
  interval2.addElement(new Integer(N/3+1));
  interval2.addElement(new Integer(2*N/3))
  mail msg2=new mail(interval2,adresse2,port2);
  sendMessage(msg2);
  Vector interval3=new Vector();
  interval3.addElement(new Integer(2*N/3+1));
  interval3.addElement(new Integer(N+1))
  mail msg3=new mail(interval3,adresse3,port3);
  sendMessage(msg3);
}
public void haveMessage(Mail m){
  // récupérer les produits des agents multiplicateurs
  Integer reponse=(Integer)m.message;
  // calcul intermédiaire de N!
resultat *=reponse.intValue();
  nbReponse++;
  if (nbReponse==3){ // dernière  réponse
    System.out.println("le resultat de N factorielle est:" resultat);
    exit(0);
  }
}
public static void main(String args[]){
  int valeur=(Integer.parseInt(args[0])).intValue();
  String a1=args[1];
  int p1=(Integer.parseInt(args[2])).intValue();
  String a2=args[3];
  int p2=(Integer.parseInt(args[4])).intValue();
  String a3=args[5];
  int p3=(Integer.parseInt(args[6])).intValue();
  agentFactorielle Nfacto = new agentFactorielle(valeur,a1,p1,a2,p2,a3,p3);
  agentNfacto.start();
}
```

```
}
//*********************************************************
//* Le code source de la classe agentMultiplicateur.class *
//*********************************************************
import java.agentCommunicant
public class agentMultiplicateur extend agentCommunicant{
public void agentMultiplicateur(int port){
  openPort(port); // réserver un port de communication
  // traitement des messages dès leur réception
  setMailInMailBox(false);
}
public void haveMessage(mail msg){
  // récupérer l'intervalle d'entiers à multiplier
  Vector interval=(Vector)msg.message;
  int N1=((Integer)interval.elementAt(0)).intValue();
  int N2=((Integer)interval.elementAt(1)).intValue();
  // calcul du produit
  int resultat=N1;
  for(int i=N1+1;i<N2;i++)  resultat *= i;
  mail resultatMsg=new mail(new Integer(resultat),msg.from);
  // renvoyer le résultat à l'agentFactorielle
  sendMessage(resultatMsg);
}
}
```

L'agent multiplicateur (la classe `agentMultiplicateur`) est considéré comme étant un agent serveur et l'agent superviseur (la classe `agentFactorielle.class`) comme étant un agent client. Pour que l'agent client puisse profiter des services des agents serveurs, ce dernier doit connaître l'adresse de chacun de ces derniers. Pour démarrer cette application, l'utilisateur doit lancer trois fois l'exécution de la classe `agentMultiplicateur` sur une même ou sur plusieurs machines distinctes. La commande pour exécuter la classe `agentMultiplicateur` est la suivante:`java agentMultiplicateur.class numPort`.

Dans le cas où la classe `agentMultiplicateur` est exécutée plusieur fois sur la même machine, les numéros de port passés en paramètre doivent être distincts. La commande pour exécuter la classe `agentFactorielle.class` est la suivante :

```
java agentFactorielle.class adresse1 port1 adresse2 port2 adresse3 port3
```

Les paramètres passés à la commande représentent respectivement l'adresse IP et le port de chacun des agents multiplicateurs.

4.7 Conclusion

En analysant le code source des applications présentées dans le paragraphe précédent, on voit bien l'utilité et la simplicité de l'utilisation de l'outil OCA. En effet, l'utilisateur peut faire communiquer les agents au sein d'un SMA en utilisant de simples instructions (`sendMessage`, `waitMessage`, `getNewMessage`, `etc.`). Les traitements de bas niveau pour l'envoi et la réception des messages sont transparents vis-à-vis de l'utilisateur. Cependant, grâce au fait que la classe `agentCommunicant` est dérivée de la classe `Thread` ainsi que tous ces composantes, l'outil OCA nous permet d'effectuer des traitements concurrents. Par exemple, au sein d'une même application, l'utilisateur peut créer et activer plusieurs agents communicants qui s'exécuteront en concurrence.

Chapitre 5

Application : Aspirateur multiagents

5.1 Introduction

Dans ce chapitre, nous proposons le développement d'une application multiagents en utilisant l'outil OCA. Cette application nous permettra de montrer un exemple de ce que nous pourrons faire avec l'outil OCA et son utilité. Le sujet de l'application a été fixé sur la simulation du nettoyage d'une surface, contenant des obstacles, par un ensemble de robots-aspirateurs. Nous présentons dans ce qui suit les spécifications et la mise en œuvre de cet application avec Java et OCA.

5.2 Spécification de l'aspirateur multiagents

L'aspirateur multiagents consiste à simuler le comportement d'un ensemble de robots-aspirateurs sur une surface poussiéreuse. L'objectif de ces robots est de balayer toutes les cases poussiéreuses de la surface. Pour des raisons de simplification de problème, nous considérons que l'espace de la surface à nettoyer est discret, formé par un ensemble de cases de forme carré. Le déplacement d'un aspirateur se fait alors d'une case à une autre dans un sens vertical ou horizontal [figure5.1]. Une case de la surface ne peut être que dans l'un des états suivants :

- case occupée par un obstacle ;
- case occupée par un robot-aspirateur ;
- case libre et propre ;
- case libre et poussiéreuse.

Pour simplifier encore l'application, nous considérons que la forme de la surface est rectangulaire. Un exemple de surface à nettoyer est présenté dans la figure 5.1. Nous considérons que l'état de la surface, entre autre l'état de chaque case, peut changer au cours du nettoyage. Une case libre peut être visitée plusieurs fois par un même robot ou par plusieurs robots (un seul à la fois). Si deux robots ou plus se dirigent simultanément vers une même case, le premier arrivé gardera sa position, les autres regagneront leurs cases.

Chaque robot est muni de capteurs lui permettant de déterminer le type des cases (obstacle, robot adjacent ou case libre) qui l'entourent et de détecter l'état de la case (propre ou poussiéreuse) sur laquelle il se trouve. Au début du nettoyage, les robots sont posés sur des

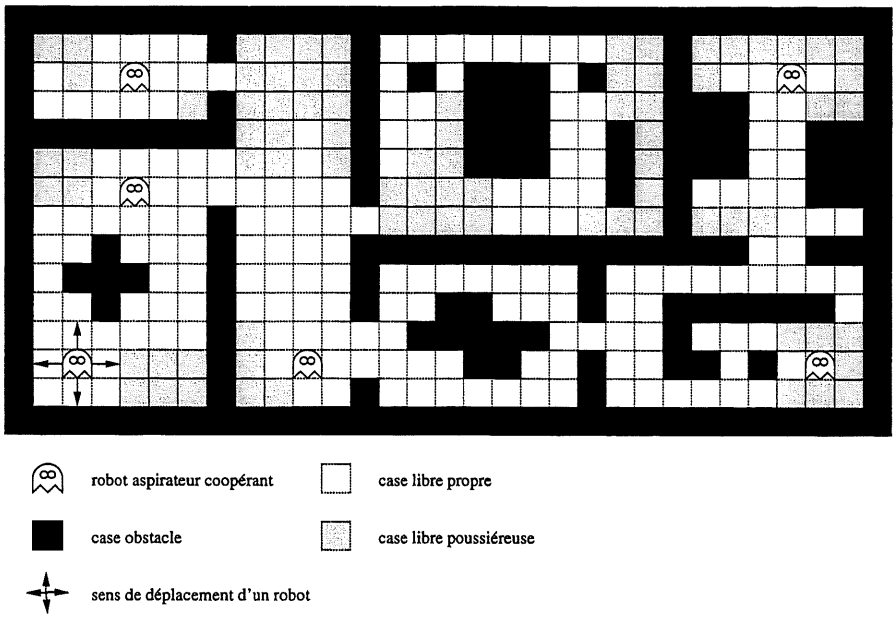

Figure 5.1 - *Plan général d'une surface à nettoyer*

cases libres quelconques. Ces derniers n'ont pas une vue globale sur la surface à nettoyer, donc ils ne connaissent pas, à priori, leur position dans la surface.

La communication entre deux robots n'est possible que lorsqu'ils se trouvent sur deux cases adjacentes. La communication permet à ces robots de collaborer pour une optimisation du nettoyage de la surface.

5.3 Structure générale du simulateur

On distingue dans cette application deux types de concept: le robot-aspirateur et la surface à nettoyer. La relation entre ces deux concepts peut être présentée sous formes d'actions

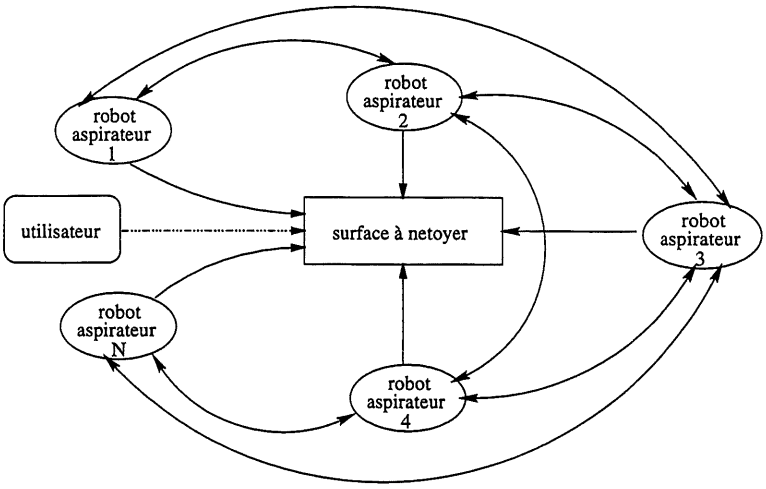

Figure 5.2 - *Architecture générale de l'aspirateur multiagents*

effectuées par l'un sur l'autre [figure 5.2] :

- Un robot-aspirateur nettoie une case de la surface ;
- Un robot-aspirateur se déplace vers une case de la surface ;
- Un robot-aspirateur détecte la présence ou l'absence de poussière sur une case ;
- Un robot aspirateur détecte la présence ou l'absence d'obstacles à son alentour.

La relation entre deux robots se réduit à un simple échange d'information concernant l'historique des cases déjà visitées par chacun des deux. Ceci est pour éviter les actions redondantes des deux côtés (une même case visitée plusieurs fois par des robots distincts) et pour accélérer le nettoyage. Dans la figure 5.2 nous présentons la structure générale du simulateur d'aspirateur multiagents.

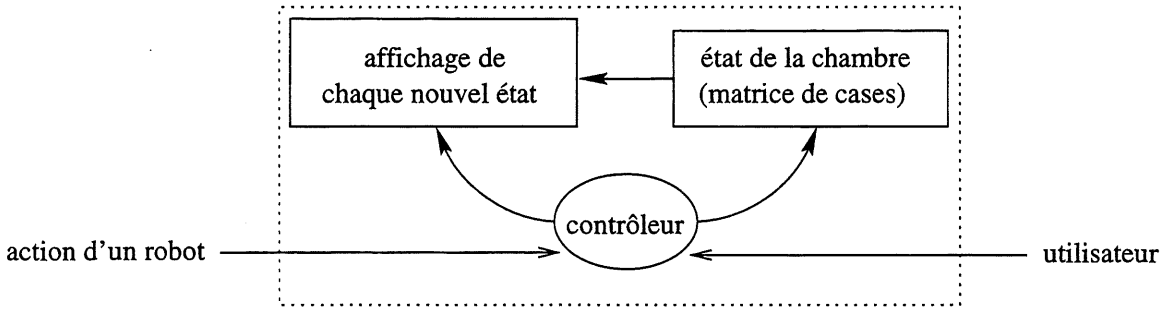

Figure 5.3 - *Modèle d'agent de type Surface*

5.4 Modélisation

Pour la modélisation de l'application, nous considérons les robots-aspirateurs et la surface à nettoyer comme étant des agents communicants.

5.4.1 Modèle d'agent communicant de type Surface

L'agent `Surface` consiste à animer l'image reflétant l'évolution d'état de la surface (l'état de chaque case ainsi que l'emplacement de chaque robot-aspirateur) durant le nettoyage. Pour rafraîchir l'image de la surface, l'agent `Surface` doit être au courant de toute action de nettoyage de case ou de déplacement de robot. L'agent surface acquit ces informations par interaction avec les agents `Aspirateurs` via la communication par envoi de message asynchrone.

Comme nous l'avons déjà précisé, l'état de la surface peut être modifié par l'utilisateur durant le nettoyage. Pour ce faire, nous permettons à l'utilisateur du simulateur d'interagir avec l'agent `Surface` pour tout ajout ou suppression d'obstacle ou de poussière sur la surface [figure 5.3].

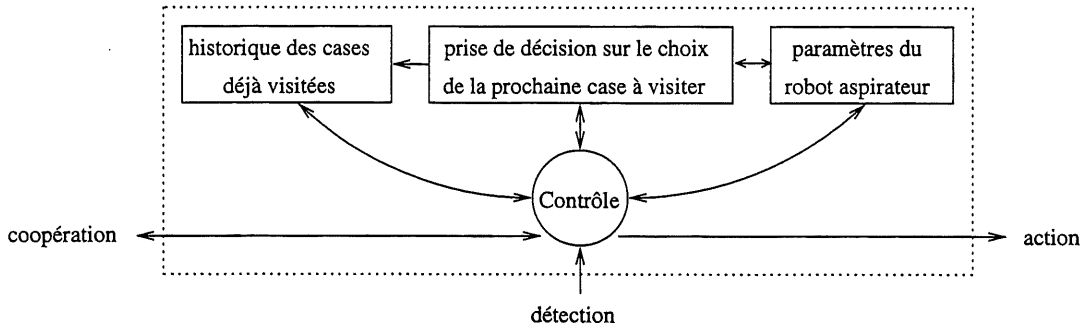

Figure 5.4 - *Modèle d'agent de type Aspirateur*

5.4.2 Modèle d'agent communicant de type Aspirateur

Le rôle d'un agent `Aspirateur` est de simuler le comportement d'un aspirateur intelligent en collaboration avec d'autres aspirateurs pour le nettoyage d'une surface. Le modèle d'un agent `Aspirateur` est présenté dans la figure 5.4.

Les principales tâches effectuées par un agent `Aspirateur` sont :

- Détection de la présence ou de l'absence de poussière sur la case sur laquelle il se trouve.
- Détection de l'absence ou de la présence d'un obstacle à son alentour.
- Nettoyage de la case sur laquelle il se trouve.
- Prendre une décision sur le choix de la prochaine case à visiter.
- Déplacement vers la case choisie.
- Communication avec un autre robot pour coopérer.

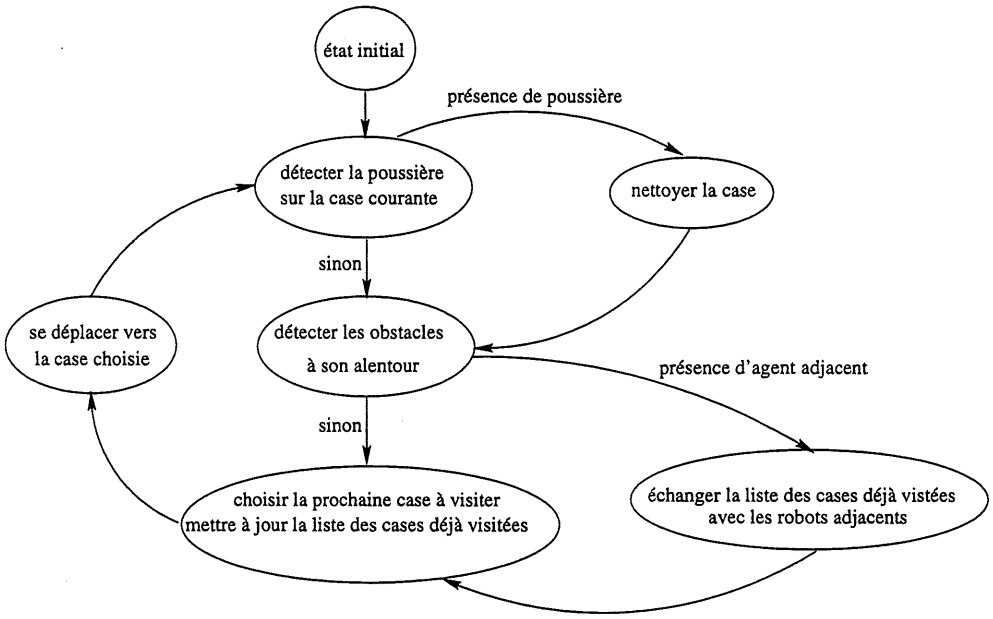

Figure 5.5 - *Description générale du comportement d'agent aspirateur*

Une description générale du comportement d'agent `Aspirateur` est présentée dans la figure 5.5. Tous les robots-aspirateurs se comportent de la même manière dans des situations identiques. Ceci n'implique pas que deux robots suiveront le même chemin s'ils partaient de la même case de départ. Pour déterminer l'impact de la coopération sur la performance de cette application, nous avons intégré dans ce modèle des paramètres permettant d'activer ou de désactiver les tâches de coopération. En pratique, le principe de base de détection d'obstacle par un robot-aspirateur est basé sur l'émission et la réflexion d'ondes infrarouges sur ces derniers. Pour la simulation, toute détection d'obstacle est exprimée par l'envoi de message (requête) de l'agent `Aspirateur` vers l'agent `Surface` suivi par un renvoi de message (réponse à la requête) de ce dernier vers l'agent `Aspirateur`. La détection de la poussière se fait avec le même principe.

Chaque agent aspirateur doit signaler tous ses mouvements de déplacement ou de net-

toyage à l'agent surface pour que ce dernier puisse rafraîchir l'image reflétant l'état de la surface avec de nouvelles dispositions des robots-aspirateurs et de nouvaux états des cases.

5.4.3 Le choix de la prochaine case

Avant chaque déplacement, l'agent `Aspirateur` doit faire un choix sur la prochaine case à visiter parmi l'ensemble des cases qui se trouvent à son alentour. En interagissant avec l'agent `surface`, l'agent `Aspirateur` peut déterminer l'état des cases à son alentour. Les cases de type obstacle sont ignorées car elles ne peuvent pas être candidates pour la prochaine case à visiter. De même pour les cases de type robot (case occupée par un robot), elles sont ignorées pour le choix de la prochaine case à visiter. Par contre, elles seront prises en considération pour la collaboration avec les robots adjacents. Le reste des cases peut être scindé en deux catégories :

1. Les cases libres déjà visitées.
2. Les cases libres jamais visitées.

Il est clair qu'une case, jamais visitée, sera plus prioritaire qu'une case déjà visitée. Parmi les cases non visitées, celles qui sont poussiéreuses seront plus prioritaires que celles qui sont propres. Comme un robot ne peut détecter la présence de la poussière que sur la case sur laquelle il se trouve, les cases non visitées auront tous la même priorité. Cependant, un robot-aspirateur peut revenir sur son choix de case, après avoir visité celle-ci, pour choisir une autre case. Ceci peut se produire si ce dernier s'aperçoit que la première case choisie était propre. Dans le cas où un robot se trouve dans la situation où toutes les cases qui l'entourent sont toutes déjà visitées, celle qui était visitée en premier temps sera la plus prioritaire. D'où

l'importance de garder les coordonnées de toutes les cases visitées avec la date à laquelle elles étaient visitées, en sachant que la case de départ du robot est de coordonnée (0,0) et visitée à l'instant 0. Chaque robot possède son propre repère temporel et spatial.

5.4.4 Coopération entre robots-aspirateurs

Durant le nettoyage, un robot-aspirateur peut effectuer des tâches redondantes. En effet, le choix de la prochaine case à visiter par un robot-aspirateur peut tomber sur une case déjà visitée par un autre aspirateur. D'où le risque que deux robots suivent le même chemin ou qu'une même case soit visitée fréquemment par plusieurs robots différents. Pour remédier à ce problème, une coopération entre les robots s'impose. Un robot doit garder en mémoire l'historique de toutes les cases qu'il a visitées et celles visitées par d'autres robots. Pour ce faire, à chaque fois que deux robots se trouvent sur deux cases adjacentes, ces derniers échangent leur liste des cases visitées par chacun. Avec ce principe, une case non visitée par un aspirateur A sera plus prioritaire qu'une autre case non visitée par le même aspirateur A, mais déjà visité par un autre aspirateur B. Ceci, diminuera la probabilité de passage par une même case plus qu'une fois. Ceci implique un nettoyage plus rapide et non redondant. On voit bien l'impact de la coopération sur l'amélioration de performance de l'aspirateur multiagents. Les agents-aspirateurs coopèrent via la communication par envoi de messages. L'architecture de communication proposée est celle du réseau en étoile. Tous les messages échangés entre les agents-aspirateurs transitent par l'agent `Surface`. Nous avons choisi cette architecture pour faciliter l'intégration des robots-aspirateurs dans le système de simulation.

5.5 Mise en oeuvre

Pour la mise en oeuvre de cette application, nous avons créé deux types de classe dérivée de la classe `agentCommunicant` :

- La classe `Surface` : Cette classe est une implémentation de l'agent `Surface`. Comme c'est l'agent `Surface` qui est sollicité par les agents `Aspirateurs`, ce dernier sera considéré comme étant un agent serveur. Dans le constructeur de cette classe, on trouve un appel de la fonction `openPort` pour la réservation du port de communication et pour répondre aux connexions externes effectuées par les agents `Aspirateurs`. L'état de la surface est représenté par une matrice à deux dimensions. Chaque élément de cette matrice correspond à l'état d'une case de la surface. Cette matrice servira par la suite comme une entrée pour l'animation de l'image reflétant l'état de la surface.

  Le traitement des messages envoyés par les agents aspirateurs, se fait directement après leur réception. C'est pourquoi nous avons activé dans cette classe le mode de traitement des messages dès leur réception avec la fonction `setMailInMailBox(false)`. Pour analyser et répondre aux requêtes envoyées par les agents aspirateurs, nous avons réécrit la fonction `haveMail` héritée de la classe mère agentCommunicant. L'information utile échangée entre les différents agents est un objet de type `request`. Cette requête est constituée par deux champs : Le premier champ, de type `String`, identifie le type de la requête. Le deuxième champ, de type `Object`, représente les paramètres de la requête. Le type du deuxième champ varie d'une requête à une autre. Tous les mouvements et les actions effectués par les agents `Aspirateurs` sont visualisés graphiquement par cette classe. Dans la figure 5.6 nous présentons un exemple d'affichage visualisé par la classe `Surface`.

- La classe `Robot` : Cette classe représente l'agent `Aspirateur`. La boucle principale de cette classe (la fonction `makeClean`) est une alternance de :
  - Détection : toutes les actions de détection de poussière ou de cases qui entourent un agent sont exprimées par l'envoi de requêtes vers la classe `Surface` suivi par une attente des messages de réponse.
  - Décision : les messages reçus (réponses aux requêtes envoyées) sont analysés et pris en considération pour le choix de la prochaine case à visiter et pour mettre à jour les connaissances de l'agent `Aspirateur`.
  - Action : le passage à l'action consiste à se déplacer vers la case choisie (une requête envoyée à la classe `Surface` pour signaler la nouvelle position du robot) ou à une coopération avec un autre robot via un échange de messages représentant l'historique des cases visitées.

La performance d'une telle application est calculée en fonction du nombre de déplacements effectués par chaque agent pour nettoyer toutes les cases poussiéreuses de la surface. La performance de l'aspirateur multiagents est définie comme étant le maximum des nombres de déplacements effectués par les agents `Aspirateurs`.

5.6 Tests et analyses

Pour les tests, nous avons fixé sept scénarios de nettoyage :

1. nettoyage d'une surface avec un seul robot-aspirateur ;
2. nettoyage d'une surface avec six robots-aspirateurs bien dispersés et sans coopération;

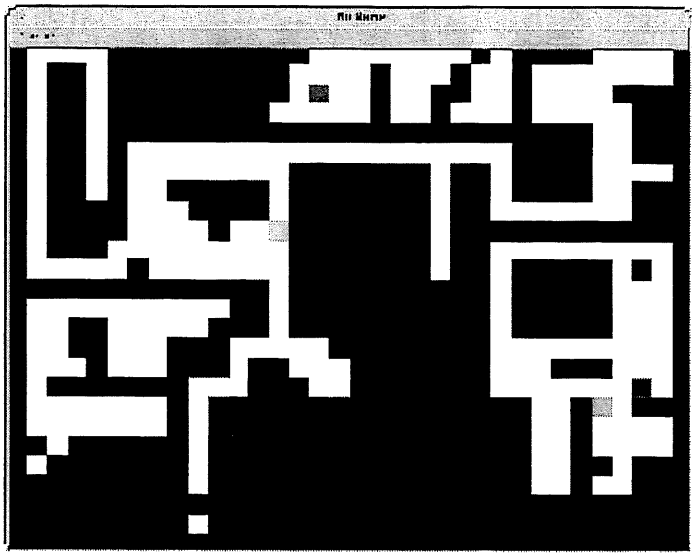

Figure 5.6 - *L'état de la surface durant le nettoyage*

3. nettoyage d'une surface avec six robots-aspirateurs bien dispersés et avec coopération;
4. nettoyage d'une surface avec dix robots-aspirateurs bien dispersés et sans coopération;
5. nettoyage d'une surface avec dix robots-aspirateurs bien dispersés et avec coopération;
6. nettoyage d'une surface avec cinquante robots-aspirateurs bien dispersés et avec coopération;
7. nettoyage d'une surface avec dix robots-aspirateurs non dispersés et avec coopération.

Il est évident que la performance du nettoyage de la surface dépendra de l'emplacement initial des robots-aspirateurs dans celle-ci. En effet, plus les robots sont dispersés dans la surface, plus la performance du nettoyage augmente. Pour cela, nous choisissons des emplacements initiaux presque identiques pour les différents tests, pour que nous puissions interpréter les résultats par la suite. De même pour l'état et la forme initiale de la surface, ils seront identiques pour les différents tests.

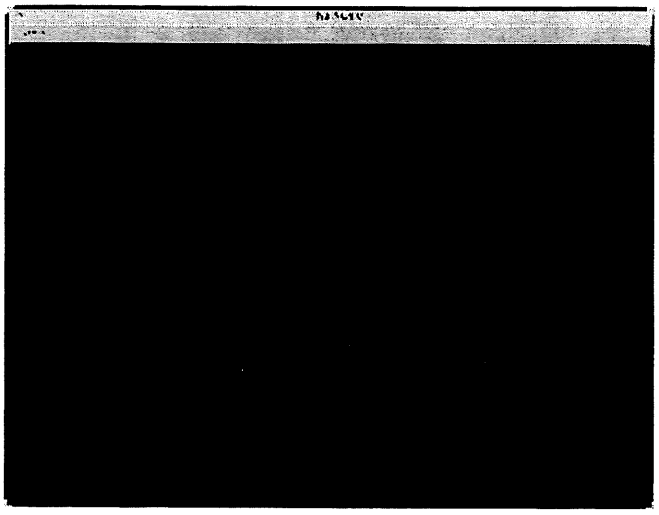

Figure 5.7 - *L'état initial de la surface*

Les agents aspirateurs seront exécutés sur des environnements hétérogène (Unix/Sun et win95/Pc). L'état initial de la surface est présenté dans la figure 5.7. Les résultats de ces tests sont décrits dans le [tableau 5.1].

Si nous comparons le premier test par rapport aux autres tests nous remarquons bien l'impact du nombre des robots-aspirateurs sur la performance du nettoyage. En comparant les tests 2 et 3 ou les tests 4 et 5, nous remarquons le rôle important que joue la coopération des robots-aspirateurs pour augmenter la performance du nettoyage multiagents. Le dernier test nous donne une idée sur l'influence de la dispersion initiale des robots dans la surface sur les résultats. C'est pourquoi nous avons choisi presque les mêmes dispersions pour les tests 2, 3, 4 et 5.

Tests	Nombre de robots	Coopération	Dispersion	Performance
Scénario 1	1	-	-	887
Scénario 2	6	sans	bien dispersés	442
Scénario 3	6	avec	bien dispersés	243
Scénario 4	10	sans	bien dispersés	259
Scénario 5	10	avec	bien dispersés	206
Scénario 6	50	avec	bien dispersés	89
Scénario 7	10	avec	non dispersés	471

TABLEAU 5.1 - *Tableau comparatif des résultats de tests des différents scénarios*

5.7 Conclusion

Le développement de cette application nous a permis de mettre en valeur l'outil OCA que nous avons mis en oeuvre. En effet, nous avons pu créer une application multiagents sans difficulté et avec un minimum de code. Le code source de cette application est disponible dans la partie annexe. Les résultats des tests de la simulation n'ont présenté aucune anomalie de fonctionnement ou d'exécution. Ce qui prouve en partie le bon fonctionnement de l'outil OCA.

D'un autre côté, le sujet de l'application nous paraît de plus en plus intéressant. En effet, les spécifications de cette application pourront être plus élaborées (degré de liberté plus élevé pour les robots, une surface de balayage non discrète, etc.) pour s'approcher plus de la réalité et pour rendre l'application plus intéressante.

Chapitre 6

Conclusion

6.1 Bilan du travail

L'objectif principal de l'outil OCA était de simplifier la tâche des programmeurs de SMA pour la mise en œuvre d'agent communicant. Cet outil a été conçu comme étant un modèle d'agent communicant intégrant les différentes fonctionnalités permettant à un agent de communiquer avec son environnement sans difficultés. Les principales fonctions se résument à: l'envoi de message, la réception de message, la diffusion de message, le traitement des événements externes, la gestion d'une boîte aux lettres et l'attente de messages etc. Parmi les fonctions intégrées dans cet outil, on trouve celles qui étaient inspirées de la littérature multiagents et d'autres que nous avons rajoutées pour améliorer et simplifier plus les tâches de communication d'un agent. Comme Java, le langage de programmation choisi pour la mise en œuvre de OCA, est portable sur plusieurs plates-formes distinctes (Windows 95/NT, Unix, OS/2, Mac), l'outil OCA sera à son tour portable sur ces mêmes plates-formes. Nous

rappelons aussi que OCA se présente sous forme d'une librairie de classes développée en Java. L'utilisation de cet outil est simple. En effet, elle consiste à : dériver la classe principale `agentCommunicant`, utiliser les méthodes prédéfinies de cette classe, redéfinir les méthodes de traitement des événements et d'en rajouter d'autres fonctions relatives à l'application multiagents.

6.2 Application

Pour l'illustration de l'outil OCA, nous avons conçu et développé une application multiagents en Java et OCA. La mise en œuvre de cette application était rapide et sans difficultés. L'application en question consistait à simuler le comportement d'un ensemble de robots-aspirateurs durant le nettoyage d'une surface avec des obstacles. Cette application nous a permis de toucher de près les concepts multiagents. Ainsi, la raison pour laquelle nous avons simplifié le sujet de cette application était la courte durée de temps réservée aux tests de l'outil OCA. En effet, nous rappelons que le but de développement de cette application était de présenter un exemple d'utilisation de l'outil OCA et de souligner l'utilité et le bon fonctionnement de l'outil. Cependant, le sujet de cette application avec des spécifications plus approfondies serait plus intéressant et pourrait être proposé dans le cadre d'un autre projet de recherche.

6.3 Perspectives

L'utilisation de l'outil OCA, pour le développement de l'outil EGSMA, consiste à reprendre la structure de OCA et développer la partie ZRCC présentée dans la figure 4.1.

Cette partie comprend les formalismes de représentation du savoir-faire, de croyance, du contrôle, et de l'expertise d'agent. Le développement de cette partie peut nécessiter une mise à jour de l'outil OCA.

Il est bien évident que l'outil OCA pourrait être enrichi par de nouvelles fonctionnalités. Nous prévoyons ainsi dans le futur de :

- Intégrer des protocoles de communication dans l'outil OCA. Nous avons déjà souligné dans le chapitre 2 le rôle des protocoles de communication qui vise à structurer et réduire les échanges de messages entre agents au sein d'un système multiagents. QKML est le protocole de communication le plus utilisé dans la littérature. Ceci nous à encouragé à envisager dans le future, dans le cadre d'un autre projet de recherche, l'intégration de ce protocole dans l'outil OCA.

- Intégrer des nouvelles fonctions pour la mobilité d'agent. Nous avons vu dans l'outil `Voyager`, présenté dans le deuxième chapitre, que les agents peuvent migrer d'un système à un autre en continuant leur exécution sur le nouveau système. Comme `Voyager` est développé en Java, de même pour l'outil OCA, ce dernier pourrait être utilisé pour rendre la classe `agentCommunicant` de l'outil OCA une classe mobile.

BIBLIOGRAPHIE

[1] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large scale parallelism. In *Conference On Foundations of Software Technology and Theoretical Computer Science*, volume 5, pages 19–41, 1985.

[2] E. A. Ashcroft, M. Clint, and C. A. R. Hoare. Remarks on program proving: Jumps and functions. *ACM Transactions on Programming Languages and Systems*, 6:317–318, 1976.

[3] R. Axelrod. *Structure of Decision: The cognitive maps of Political Elite.* Princeton University Press, 1976.

[4] D. Badouel and P. Cointe. Java: Quelque part entre smalltalk et c++. *L'Objet*, 2(5):15–22, June 1996.

[5] H. S. Becker. Notes of the concept of commitment. *American Journal of Sociology*, 66:32–40, 1960.

[6] O. Boissier, Y. Demazeau, and S. Berthet. Knowing each other better. *DAI Workshop*, 11:1–20, 1992.

[7] A. H. Bond. Commitment: A computational model for organizations of cooperating intelligent agents. In *Conference on Office Information Systems*, pages 21–30, 1990.

[8] A. H. Bond and L. Gasser. Readings in distributed artificial intelligence, 1988.

[9] J-P. Briot. Modélisation et classification de langages de communication concurrente à objet : l'expérience actalk. In *Langages et Modèles à Objets*, pages 103–125, Octobre 1994.

[10] S. Cammarata, D. McArthur, and R. Steeb. Strategies of cooperation in distributed problem solving. In Karlsruhe, editor, *International Joint Conference on Artificial Intelligence*, pages 767–770, August 1983.

[11] M. K. Chang and C. C. Woo. Snap: A communication level protocol for negotiations. *MAAMAW*, 3, 1991.

[12] V. Chevrier. Coordination et structuration des échanges par négociation dans les systèmes multiagents. In *Journée Systèmes Multiagents PRCGDR Intelligence Artificielle*, Décembre 1992.

[13] S. E. Conry, R. A. Meyer, and V. R. Lesser. Multi-stage negotiation in distributed planning, 1988.

[14] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. In *Artificial Intelligence*, volume 20, pages 63–109, 1983.

[15] R. Davis and R.G Smith. Framework for cooperation in distributed problem solving. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 61–70, 1990.

[16] E. H. Dufee, V. R. Gasser, and D. D. Korkill. Coherent cooperation among communicating problem solvers. *IEEE Transactions on computers*, C-36:1275–1291, 1987.

[17] E. H. Dufee and V. R. Lesser. Using partial global plans to coordinate distributed problem solving. In *IJCAI*, volume 10, pages 875–833, 1987.

[18] L.D Erman and V.R Lesser. Distributed interpretation : A model and experiment. In *IEEE Trans Computer*, volume 29, pages 1144–1166, 1980.

[19] J. Ferber. *Conception et Programmation par Objets.* Hermes, 1990.

[20] J. Ferber and M. Ghallab. Problématique des univers multi-agents intelligents. *Journées nationales sur Intelligence Artificielle PRC-GRECO*, pages 295–320, 1988.

[21] T. Finin, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, R. Pelavin, S. Shapiro, and C. Beck. Specification of the kqml agent-communication language. Technical report, The DARPA Knowledge Sharing Initiative External Interfaces Working Group, 1993. Is also available at http://www.csee.umbc.edu/kqml.

[22] H. Robert Frost. *Java(tm) Agent Template.* Enterprise Integration Technologies, Inc, 1996. Is also available at http://cdr.stanford.edu/ABE/documentation/.

[23] G. Gaspar. Communication and belief changes in a society of agents : Towards a formal model of autonomous agents. *Decentralized AI*, 2, 1991.

[24] D. Gauvin. Un environnement de programmation orientée agent, Mars 1995.

[25] M. L. Ginsberg. Decision procedures, 1987.

[26] J. Gosling and H. McGilton. The java language environment. Technical report, Sun Microsystems, Inc, October 1995.

[27] Object Management Group. The common object request broker 2.0 : Architecture and specification, December 1995.

[28] Z. Guessoum. Dima: Une plate-forme de conception et de réalisation de systèmes mutiagents en smalltalk. Version électronique est valable au http://www.iicm.edu/jucs/.

[29] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *ACM Conference Principles of Distributed Computing*, volume 3, pages 50–61, 1984.

[30] C. Hewitt and W.A Kornfeld. The scientific community metaphor. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 11, pages 24–33, 1981.

[31] M. N. Huhns. *Distributed Artificial Intelligence.* Pitman Learning Inc, 1987.

[32] K. Konologie. Agents with attitudes. Invited talk/panel at the AAAI, 1991.

[33] S. Labidi and W. Lejouad. De l'intelligence artificielle distribué aux systèmes multiagents. Technical Report 2004, Unité de recherche INRIA Sophia-Antipolis, Août 1993.

[34] V.R Lesser and D.D Corkill. The use of meta-level control for coordination in a distributed problem solving network. In *International Joint Conference Artificial Intelligence*, volume 8, pages 748–756, 1983.

[35] V.R Lesser and E.H Durfee. Using partial global plans to coordinate distributed problem solvers. In *International Joint Conference Artificial Intelligence*, volume 10, pages 875–883, 1987.

[36] T. Maruichi, M. Ichikawa, and M. Tokoro. *Decentralized AI*, chapter Modelling Autonomous Agents and their Groups, pages 215–134. Elsevier Science, 1990.

[37] A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

[38] OBJECTSPACE. *ObjectSpace Voyager Version 2.0.0 Production Release*, 1996. Is also available at http://www.objectspace.com.

[39] C. Pegard. *Coordination de robots mobiles Autonomes, application aux chantiers automatisés.* PhD thesis, Université de Picardie, 1988.

[40] J. R. Searle. *Intentionality: In Essay in the philosophy of mind.* Cambridge University Press, 1983.

[41] J. R. Searle. *Intensions in Communication.* MIT Press, London, 1990.

[42] J. Searlmen, R.A Mayer, and S. Conry. A shared knowledge base for independent problem solving agents. In *IEEE Expert Systems in Government Symposium*, 1988.

[43] S. S. Sian. Adaptation based on cooperative learning in multi-agent systems. In *Decentralized Artificial Intelligence*, volume 2, 1991.

[44] Sun Microsystems, Inc. *JDK: The Java Developers Kit*, 1998. is also available at http://java.sun.com/products/JDK/index.html.

[45] Sun Microsystems, Inc. *RMI: Remote Method Invocation*, 1998. Is also available at http://java.sun.com/products/jdk/rmi.

[46] Tuomela, Raimo, Kaarl, and Miller. We intentions. *Philosophical Studies*, 53:367–389, 1988.

[47] F. Vernadat and P. Azéma. Prototypage de systèmes d'agents communicants. *Journée Systèmes Multiagents PRC-GDR Intelligence Artificielle*, décembre 1992.

[48] G. Vernazza, S.B Serpico, C. Regazzoni, and S. Dellepiane. Application independent knowledge-based framework for complex image recognition. In *International Conference Analysis and Processing*, volume 5, 1989.

[49] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming.* The MIT Press, 1987.

ANNEXE

Le code source de la classe Chambre.class

```
import java.awt.*;
import java.io.*;
import java.util.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseMotionAdapter;
import java.agentCommunicant;
import java.mail;
class ChambreApplet extends Frame {
// Cette classe, dérivée  de la classe Frame, consiste à afficher et animer
// l'état de la surface à  nettoyer. Nous  présentons pas le code de cette
//  classe. En effet, le code de cette classe n'a aucun rapport avec l'outil
// OCA.
}
public class Chambre extends agentCommunicant {
private Hashtable canaux= new Hashtable();
private ChambreApplet app;
public Chambre (ChambreApplet ap){  app=ap;  }
// redéfinition de la fonction haveMessage pour traiter les messages reçus
public void HaveMessage(mail Msg){
   request req=(request)Msg.message;
```

```
String cmd=req.commande;
if (cmd.equals("getFistrPosition")) {
   String robot= (String)req.attribut.elementAt(0);
   canaux.put(robot,new Integer(Msg.canal));
   request outReq=app.getFirstPosition(robot);
   mail OutMail=new mail(0,-1,Msg.canal,outReq);
   sendMessage(OutMail);
}
else
if (cmd.equals("getContourCase")) {
   request outReq;
   String robo=(String)req.attribut.elementAt(0);
   int xy[]=(int[])req.attribut.elementAt(1);
   int direction[]=(int[])req.attribut.elementAt(2);
   int res=app.moveRobot(robo,xy);
   if (res < 100){
      outReq.commande="takeContourCase";
      outReq.attribut=app.getContourCase(robo,direction,xy);
   }
   else {
      outReq.commande="robotTouche";
      outReq.attribut.addElement(res);
   }
   mail OutMail=new mail(0,-1,Msg.canal,outReq);
   if (!sendMessage(OutMail))
      System.out.println("Problem d'emission de la chambre");
}
else
if (cmd.equals("makeCooperation")) {
   String from=(String)req.attribut.elementAt(0);
   String toRobot=(String)req.attribut.elementAt(1);
   Vector caseVisite=(Vector)req.attribut.elementAt(2);
   request outReq=new request();
   outReq.commande="makeCooperation";
```

```
        outReq.attribut.addElement(from)
        outReq.attribut.addElement(caseVisite)
        Integer canal=(Integer)canaux.get(to);
        mail OutMail=new mail(0,-1,canal,outReq);
        if (!sendMessage(OutMail))
           System.out.println("Problem d'emission de la chambre");
    }
    else  System.out.println("Commande inconnue ........! ");
}
public void start(int portChambre){
    OpenPort(portChambre);
}
public static void main(String args[]) {
    ChambreApplet app=new ChambreApplet();
    app.init();
    Chambre chambre=new Chambre(app);
    chambre.start(9000);
}
}
```

Le code source de la classe Robot.java

```
//*****************************************
//* Le code source de la classe Robot.class *
//*****************************************
import java.agentCommunicant;
import java.mail;
import java.util.*;
class Robot extends agentComunicant{
private String hostSurface;
private int portSurface;
private int surface; // numéro du canal établi avec l'agent surface
private String roboName; // l'identificateur de l'agent robot
```

```
private int[] direction={-1,0}; // direction de l'agent robot
private int[] xy={0,0,0}; // coordonnées de l'agent robot
private int[] Prevxy={0,0,0}; // coordonnées précédentes de l'agent robot
private boolean haveJob=true; // L'état du robot
private Hashtable caseVisite; // table des cases visitées par le robot
// table des cases visitées par d'autres robots
private Hashtable caseVisiteParAmis;
// la liste des cases échangées avec d'autres robots
private Hashtable caseEnvoyeAmis;
public void start(String name,String hostCh,int portCh){
   roboName=name;
   portSurface=portCh;
   hostSurface=hostCh;
   caseVisite=new Hashtable();
   caseVisiteParAmis = new Hashtable();
   caseEnvoyeAmis = new Hashtable();
   // déterminer la position initiale du robot aspirateur
   xy=getFistrPosition();
   Prevxy=xy;
   // commencer le nettoyage
   makeClean();
}
// la boucle principale de l'agent robot aspirateur
public void makeClean(){
while (haveJob){
   // détection du contour
   int[] contour=getContourCase(xy);
   if (contour!=null){
      // mise à jour de la liste des cases visitées
      addCaseVisite(xy);
      // choix de la prochaine case à visiter
      int pos[]=nextCase(contour);
      ChangeDirection(pos);
      Prevxy=xy;
```

```
      xy=pos;
   }
   else {
      // ici, c'est le cas ou deux robots se dirigent vers la même case
      addCaseVisite(xy);
      xy[2]--;
      // retourner à la case précédente
      xy=Prevxy;
   }
}
}
// cette fonction permet d'arreter le traitement à la suite d'une détection
// de rupture du canal établis avec l'agent surface
public void dialogueIsClosed(int canal){
haveJob=false;
}
// cette fonction permet d'envoyer une requête à l'agent surface pour la
// détection du contour
public int[] getContourCase(int pos[]){
request req =new request();
req.commande="getContourCase";
req.attribut.addElement(roboName);
req.attribut.addElement(pos);
req.attribut.addElement(direction);
Mail requeteContour=new Mail(0,-1,surface,req);
if (sendMessage(requeteContour)){
   do{
      Mail answerContour=waitMessage(surface);
      request reponse= (request)answerContour.message);
      String cmd =contour.commande;
      if (cmd.equals("takeContourCase")){
         int cases[]=reponse.attribut.elementAt(1)
         return cases;
      }
```

```
      else
      if (cmd.equals("makeCooperation")){
         String from=reponse.attribut.elemenrAt(1);
         int[] caseVisit= reponse.attribut.elemenrAt(2);
         updateCaseVisiteParAmis(caseVisite);
      }
      else
      if (cmd.equals("robotTouche")){
         int robot= reponse.attribut.elemenrAt(1);
         makeCooperation(robot);
         return null;
      }
      else System.out.println("Commande inconnu !");
   } while (true);
}
else System.out.println("Problème d'émission !");
return null;
}
// cette fonction détermine la prochaine case à visiter
public int[] nextCase(int[] contour){
Vector contourLibre=new Vector(); // les cases libre du contour
Vector contourDejaVisite=new Vector(); // les cases du contour déjà visitées
nt x=xy[0];
int y=xy[1];
Integer deja[]=null;
Integer min[]=null;
Integer d[]=direction;
// trier les cases du contour suivant un ordre de priorité
for(int i=0;i<4;i++){
   int pos[]={x+d[0],y+d[1],xy[2]+1};
   d=rotation(d);
   if (contour[i]==1) continue; // ignorer les obstacles
   deja=dejaVisite(pos);
   if (deja==null) contourLibre.addElement(pos);
```

```
   else contourDejaVisite.addElement(deja);
}
if (contourLibre.size()!=0) return (int[])contourLibre.firstElement();
else{
   if (contourDejaVisite.size()!=0){
      // trier les cases visitées suivant la date de passage
min=(int[])contourDejaVisite.firstElement();
      for(int i=1;i<contourDejaVisite.size();i++){
         int pos[]=(int[])contourDejaVisite.elementAt(i);
         if(pos[2]<min[2]) min=pos;
      }
   }
}
if (min[2]==xy[2]+1) return xy;
else {
   min[2]=xy[2]+1;
   return min;
}
}
// cette fonction permet d'echanger avec un autre agent les cases visitées
public void makeCooperation(String withRobot){
int Case[];
Vector ListCaseVisiteToSend=new Vector();
int i=1;
for (Enumeration e = caseVisite.elements() ; e.hasMoreElements() ;i++){
   Case=(int[])e.nextElement();
   String code=Case[0]+"-"+Case[1]+"-"+withRobot;
   int c[]=(int[])caseEnvoyeAmis.get(code);
   if (c==null){
      String s=Case[0]+"-"+Case[1]+"-"+Case[2]+"@";
      ListCaseVisiteToSend.addElement(Case);
      caseEnvoyeAmis.put(code,Case);
   }
}
```

```
if (ListCaseVisiteToSend.size()==0) return;
request req = new request(); req.commande= "makeCooperation";
req.attribut.addElement(roboName);
req.attribut.addElement(withRobot);
req.attribut.addElement(ListCaseVisiteToSend);
Mail makeCooper=new Mail(0,-1,surface,request);
if (sendMessage(makeCooper)) System.out.println("cooperation is done");
else System.out.println("cooperation is failed");
}
// cette fonction permet d'envoyer une requête à l'agent surface pour
// déterminer la position initiale
public int[] getFistrPosition(){
int[] rep={0,0,0};
// établir un dialogue avec l'agent surface
surface = conectAgent(hostSurface,portSurface);
request req=new request();
req.commande="getFistrPosition";
req.attribut.addElement(roboName);
mail requeteXY=new mail(0,-1,surface,req);
sendMessage(requeteXY);
do{
   mail answerXY=waitMessage(surface);
   request req=answerXY.message;
   String cmd=req.commande;
   if (cmd.equals("getFistrPosition")){
   int r[]=(int[])req.attribur.elementAt(1);
rep[0]=r[0];
rep[1]=r[1];
   rep[2]=0;
   return rep;
}
} while (true);
}
// cette fonction permet de déterminer si une case était déjà visitées ou pas
```

```
public int[] dejaVisite(int x,int y){
int c[]= (int[])caseVisite.get(x+"-"+y);
if (c==null) return (int[])caseVisiteParAmis.get(x+"-"+y);
else return c;
}
// fonction pour mettre à jour les cases déjà visitées
public void addCaseVisite(int pos[]){
String code=pos[0]+"-"+pos[1];
caseVisite.remove(code)==null) ;//System.out.println("case visité:"+p(pos));
caseVisite.put(code,pos);
}
// fonction pour mèttre à jour les cases déjà visitées par d'autres robots
public void updateCaseVisiteParAmis(Vector List){
int pos=List.indexOf("@");
if (pos!=-1){
   String s=List.substring(0,pos);
   int Case[]={0,0,0};
   Case[0]=List.elementAt(0)[0];
   Case[1]=List.elementAt(0)[1];
   Case[2]=0;
String code=Case[0]+"-"+Case[1];
caseVisiteParAmis.remove(code);
caseVisiteParAmis.put(code,Case);
   List.
updateCaseVisiteParAmis();
}
else return;
}
public void ChangeDirection(int pos[]){
direction[0]=pos[0]-xy[0];
direction[1]=pos[1]-xy[1];
}
int[] rotation(int dir[]){
if (dir[0]==-1) {dir[0]=0;dir[1]=-1;} else
```

```
if (dir[1]==-1) {dir[0]=1;dir[1]=0;}  else
if (dir[1]==1) {dir[0]=-1;dir[1]=0;}  else {dir[0]=0;dir[1]=1;}
return dir;
}
// programme principal pour créer et activer l'agent robot aspirateur
public static void  main(String args[]) {
int port=9000;
String host="pollux";
String numeroRobot="100";
if (args.length!=0) {
   numeroRobot=args[0];
   host=args[1];
}
Robot leRobot=new Robot();
leRobot.start(numeroRobot,host,port);
}
}
```